The Pragmatic Programmers

Explore It!

Reduce Risk and Increase Confidence with Exploratory Testing

Elisabeth Hendrickson

Edited by Jacquelyn Carter

Early Praise for *Explore It!*

Reading this book taught me new skills and heuristics; but even better, it helped me channel my tester "spidey sense" more creatively and usefully. I keep this book handy at all times and occasionally do one of the practice sessions to keep my awareness keen. *Explore It!* helps me make sure our customers and our company get real value from our software. It'll help you too.

➤ **Lisa Crispin**
Coauthor with Janet Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*

Explore It! starts with a bang. Elisabeth catches your imagination and has filled the book with practical ideas for exploring everything from your typical GUI scenarios to testing ideas (requirements), and she even includes suggestions for programmers on how to explore low-level code. This book should be on every development team member's desk, not only testers. It is the book I carry with me whenever I introduce exploratory testing to development teams.

➤ **Janet Gregory**
Coauthor with Lisa Crispin, *Agile Testing: A Practical Guide for Testers and Agile Teams*

At Moolya, we were wondering how to get newbie testers started with exploratory testing. *Explore It!* appears to be timely and useful to the problem we have in hand.

➤ **Pradeep Soundararajan**
Founder, Moolya Software Testing Private Limited

There's a hidden secret to *Explore It!* While attempting to capture the nuance and depth of exploratory testing, Elisabeth Hendrickson has managed to write one of the best books on test design I've ever read. Good test design comes from good test ideas, and this book is overflowing with both great test ideas and pragmatic stories to back them up.

➤ **Alan Page**
Microsoft

Elisabeth Hendrickson has been working on a book ever since I met her many years ago. This book was worth waiting for! *Explore It!* explores exploratory testing, a topic that desperately needed a well-written book. Not only is this book clear, good, and useful, but Elisabeth's stories also make it a fun read. I recommend exploring it.

➤ **Bas Vodde**
Odd-e

Explore It! is packed with a ton of practical advice on planning and executing exploratory testing, a topic that is becoming increasingly popular with Agile teams. As an introduction to the topic, the book should be interesting to anyone serious about quality on Agile projects, but it will be particularly useful to testers working with Agile teams as a reference for things to investigate. I found the chapters on various heuristics for finding interesting things to explore particularly interesting, including spotting variables, varying sequences and interactions, and exploring state transitions.

➤ **Gojko Adzic**

Elisabeth has written a book that should be on every team's bookshelf, well thumbed and frequently referenced! She explains what exploratory testing is—and very importantly, what it isn't—in an accessible and easy-to-understand style. She tells interesting stories and offers examples that explain and clarify the advice she gives. I heartily recommend this book to anyone who is involved in testing real-world software systems.

➤ **Shane Hastie**

When I saw this book announced, I knew I had to read it as soon as possible. I was amazed, delighted, and overwhelmed by the practical guidance that Elisabeth provides. This book closes the gap on exploratory testing on the recommended reading list for my ET classes.

➤ **Markus Gärtner**
Author of *ATDD by Example: A Practical Guide to Acceptance Test-Driven Development*, it-agile GmbH

Elisabeth Hendrickson's terrific new book is an essential guide not only for testers but also for the whole team. New testers will learn how to generate innumerable useful test ideas quickly. Experienced testers will gain nuance and fresh insight into techniques they've used for years, and they will learn how to apply their existing skills to help their teams in new ways. Managers and developers will learn how exploratory testing can help them stay productive and keep out of trouble before, during, and after development.

➤ **Dale Emery**
Consultant

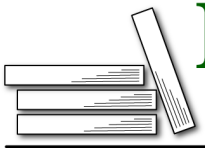
Explore It!

Reduce Risk and Increase Confidence with Exploratory Testing

Elisabeth Hendrickson

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jackie Carter (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Copyright © 2013 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-937785-02-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—September 2013

Contents

Foreword . . . . . **xi**

Acknowledgments . . . . . **xiii**

Preface . . . . . **xv**

Part I — Establishing Foundations

1. On Testing and Exploration **3**
- 1.1 Two Sides of Testing 4
- 1.2 Essential Elements of Exploratory Testing 6
- 1.3 Working in Time-Boxed Sessions 7
- 1.4 Practice Sessions 8

2. Charter Your Explorations **11**
- 2.1 Chartering the Expedition 11
- 2.2 A Simple Charter Template 13
- 2.3 Good Charters 15
- 2.4 Generating Charters 16
- 2.5 The Nightmare Headline Game 21
- 2.6 Planning Charters 24
- 2.7 Practice Sessions 24

3. Observe the Details **25**
- 3.1 But Did You See the Moonwalking Bear? 25
- 3.2 Digging Deeper 27
- 3.3 Testability and Making the Invisible Visible 30
- 3.4 Consoles and Logs 31
- 3.5 Practice Sessions 32

4. **Find Interesting Variations** 33
4.1 Variables Are Things That Vary 34
4.2 Subtle Variables, Big Disasters 36
4.3 Identifying Variables 38
4.4 Variables! They're Everywhere! 44
4.5 Practice Sessions 44

5. **Evaluate Results** 47
5.1 Never and Always 48
5.2 Alternative Resources 50
5.3 Approximations 52
5.4 Practice Sessions 55

Part II — Adding Dimensions

6. **Vary Sequences and Interactions** 59
6.1 Nouns and Verbs 60
6.2 Random Navigation 62
6.3 Personas 63
6.4 Practice Sessions 64

7. **Explore Entities and Their Relationships** 67
7.1 Recognizing Entities, Attributes, and Dependencies 67
7.2 CRUD: Create, Read, Update, Delete 70
7.3 Following the Data 73
7.4 Practice Sessions 73

8. **Discover States and Transitions** 75
8.1 Recognizing States and Events 76
8.2 Diagramming a State Model 78
8.3 Exploring with State Models 80
8.4 Changing the Representation: State Tables 82
8.5 Practice Sessions 85

9. **Explore the Ecosystem** 87
9.1 Diagramming the Ecosystem 87
9.2 Trust Boundaries 92
9.3 What If? 93
9.4 Follow the Data 95
9.5 Practice Sessions 96

Part III — Putting It in Context

10. Explore When There Is No User Interface 99
10.1 Exploring an API 100
10.2 Exploring a Programming Language 102
10.3 Exploring a Web Service 104
10.4 Characterizing Bugs 106
10.5 Practice Sessions 107

11. Explore an Existing System 109
11.1 Starting with a Recon Session 109
11.2 Sharing Observations 112
11.3 Interviewing Stakeholders to Gather Questions 114
11.4 Selecting Techniques 116
11.5 Recording Your Findings 116
11.6 What About the Dreaded Unreproducible Bugs? 117
11.7 Practice Sessions 119

12. Explore Requirements 121
12.1 Getting Into the Requirements Meeting 122
12.2 In the Requirements Meeting 125
12.3 Charter During Requirements Discussions 128
12.4 Active Reading 130
12.5 Practice Sessions 132

13. Integrate Exploration Throughout 133
13.1 Exploring as Part of Your Test Strategy 133
13.2 Paired Exploration 137
13.3 Discovering Systemic Sources of Problems 138
13.4 Estimating Exploration 140
13.5 How to Tell When You Have Explored Enough 141
13.6 Debriefing Stakeholders 142
13.7 Capturing Useful Nuggets of Wisdom 144
13.8 Practice Sessions 145

A1. Interviewing for Exploratory Testing Skills 147
A1.1 Paired Exploratory Testing as Part of an Interview 148
A1.2 Assessing the Candidates 150

A2. Test Heuristics Cheat Sheet 151
A2.1 General Heuristics 151
A2.2 Web Heuristics 156

Bibliography 157

Index 159

Foreword

A small group of people can exert influence over many by bundling innumerable decisions into a logical artifact called a program. The full nature of that influence defies a priori assessment. This leaves the influential wondering: Is there anything more we might want to change?

There is the philosophical question: How do we know what we know?

There is the economic question: When will we know enough to move forward with confidence?

There is the psychological question: How do we convince each other that one more change is needed, or is not needed?

Elisabeth Hendrickson shows us how to answer these questions through a methodical inquiry called exploratory testing. Elisabeth speaks with the authority of a successful practitioner and an experienced teacher. She has pioneered the application of exploratory testing to Agile development throughout its emergence as mainstream practice. And she is a compassionate human being who has successfully conveyed through this book the simple pleasure of working together and the delights of making discoveries that other people value.

Exploratory testing is important.

Although the approach delivers value in any kind of development, it is especially appropriate in Agile, where fast cycles and sudden changes are common. Development and testing methods share much in common. Both favor studied craftsmanship. They expect the practitioner to be alert, motivated, and capable of judgment. Both recognize that decisions are situated in a business context even when they appear highly technical and even obscure.

Exploratory testing can be learned.

Exploratory testing offers fresh approaches to traditional problems. Elisabeth sidesteps old battles by confirming what has not changed: risk and the

confidence that comes from keeping it in check. Then, as a storyteller, she works from fundamentals through to a clear description of each practice, with catalogs of tips for working well. Each chapter closes with practice sessions you can do now in your own environment.

Exploratory testing will satisfy.

Without negating the value of many specialized testing methods, exploratory testing steps forward as the most universally applicable and most obviously collaborative process team members can use to gain insight into what they've built and how others will experience it. It rewards curiosity with information that can delight on its own as well as deliver immense business value.

Ward Cunningham
Portland, Oregon

Acknowledgments

When you think about a writer at work, you may picture a solitary figure hunched over a keyboard. The reality is a little different: it really does take a whole community to produce a book.

First, my heartfelt thanks go to my editor, Jackie Carter. Jackie guided me through the entire process. There were any number of times I would have been hopelessly lost without her. What's more, Jackie has a knack for asking probing questions designed to elicit details that previously existed only in my head and reminding me I needed to write those details down. The result is a much more complete, well-rounded, and readable book. If you find the book hangs together as a cohesive whole, thank Jackie. (If you don't, blame me.)

I'm also thankful to Pragmatic Bookshelf and the publishers, Andy Hunt and Dave Thomas, for taking me on as a new author and for publishing this book. In addition, I am grateful to them for creating an ecosystem of tools that were a true joy to use. Thanks also to Molly McBeath for her attention to detail in copyediting.

I am extremely grateful to all the reviewers, as well as to the people who submitted errata during the beta. I deeply appreciate the time and attention all of them gave to the manuscript. This book would have been a pale shadow of what it is had they not offered their insight. So thank you to Anna Baik, Matthew Barcomb, David Christiansen, Simon Crutchley, Ian Dees, George Dinwiddie, Svein Håvard Djupvik, Yves Hanoulle, Paddy Healy, Harry Heckel, Glen Ivey, Amy Lightholder, Rickard Lindberg, Erik Petersen, Dave Rooney, Craig Smith, Pradeep Soundararajan, John Stevenson, Joe Strazzere, Bill Wake, Jo Webb, and Adam Yuret.

Many thanks to Erik G. H. Meade for introducing me to the Galactic Modeling Language (GML). It greatly simplified many of the diagrams in this book.

Several people have deeply influenced my thinking. Dale Emery helped me understand the significance of concepts like variables that I previously took for granted. James Lyndsay pushed me to see past traditional test design

approaches and take a more investigative approach. James Bach challenged me, and my thinking is sharper as a result. Jon Bach questioned me in a different style and honed my thinking even further. Cem Kaner taught me a tremendous amount. Moreover, Cem created the Los Altos Workshop on Software Testing (LAWST), a peer-driven forum for examining experiences and ideas. I'm grateful to have had the opportunity to participate in LAWST at a time when I sorely needed a community of practice. All the LAWST community members shaped my thinking in some way. Finally, Jerry Weinberg has been a mentor and friend, and he taught me a great deal about both testing and storytelling.

Writing a book is a time-consuming endeavor, and I am immensely grateful to my husband, Kirk Hendrickson, for his love and support, as well as to my children, Emily and Anna, for their patience with me.

Finally, my thanks to the broader community of practitioners who speak and write and share their experiences as explorers. I am inspired by you.

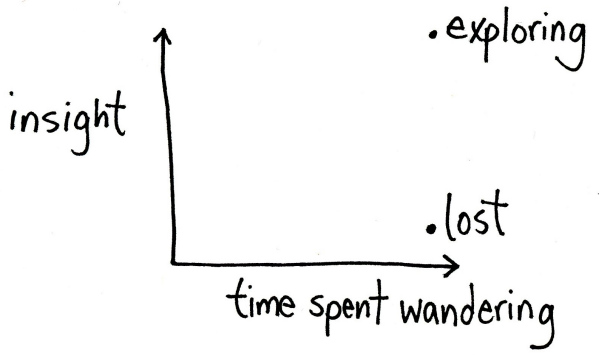
Preface

In order to truly understand anything, you have to explore it.

This is true of cities. When I travel, I always reserve at least a little time to wander the back streets and find hidden gems. I escape the tourist areas to find the restaurants where only locals go and the shops that cater to everyday needs. It is in these meanderings that I begin to understand the local culture and become acquainted with the real character of the place.

It's true of software as well. If you want to learn about the real capabilities and limitations of your software, you have to navigate off the beaten path.

Yet there is a difference between wandering at random and real exploring, as Jessica Hagy captured so eloquently in her Indexed web comic titled "Field notes."[1]



Jessica's comic carries an important word of warning. If you meander without direction or purpose, you'll spend much time wandering for precious little insight: you'll be lost.

This is a book about how to explore well. Within these pages you'll discover the essential skills and techniques of a master explorer.

1. Cartoon by Jessica Hagy, used with permission (http://thisisindexed.com/2011/10/field-notes/).

Who Is This Book For?

Because this is a book about software testing, you might think that it's for testers. It is, but not just testers. I wrote this book for anyone who has an interest in producing reliable and robust software. That includes programmers, business analysts, product managers, and even the people who usually see the software only when it's close to done: the support personnel.

If you are a tester with a firm grasp on test design techniques such as state analysis, you'll learn how to use those skills to design tests on-the-fly while exploring.

If you are a programmer with a deep understanding of the underlying technology, you'll learn how to view the software through different lenses, analyzing it from various perspectives.

If you are a business analyst or product manager, you'll learn how to vary your interactions with the software to ensure it does what you intended under a wider variety of circumstances.

If you are in support, you'll discover how to deepen your understanding of the software you're supporting. You may also pick up some tricks for replicating those hard-to-reproduce customer issues.

In short, if you have a hand in creating software, whether specifying it, programming it, verifying it, or supporting it, this book is for you.

It also doesn't matter what kind of software you work on. The techniques in this book apply to a wide range of technical contexts, from web applications to desktop applications to mobile applications to embedded real-time software to APIs and web services.

How This Book Is Organized

This book is organized into three parts:

- Part I, *Establishing Foundations*, on page 1, introduces core building blocks for becoming skilled at exploring. In this section, you'll learn how to craft charters to guide your exploration, how to observe what's really happening (hint: it's harder than it sounds), how to identify interesting variations, and how to determine what behavior should be expected when exercising software in ways no one ever thought about before.
- Part II, *Adding Dimensions*, on page 57, builds on that foundation. You'll learn how to explore by varying interactions, sequences, data, timing, and

configurations. Along the way you'll learn how analysis techniques like state modeling and data modeling can support your exploration.

- Part III, *Putting It in Context*, on page 97, brings the techniques back into the context of a software project. You'll learn how to apply the ideas in parts 1 and 2 in a variety of contexts, including exploring existing (legacy) applications and exploring software that has no user interface. You'll also learn how to share your findings, as well as how to integrate exploration into the development cycle from the very beginning.

Although you can read the chapters in any order, you will get the most value out of each part after you've mastered the concepts in the preceding parts.

Practice is essential for growing your exploration skills. To help you do that, each chapter ends with a section titled "Practice Sessions" that will help you cement the concepts in your mind and give you ideas for applying the concepts in the chapter right away.

Throughout the book, you'll find specific techniques, or heuristics. Each heuristic has a name, like "Some, None, All" or "Beginning, Middle, End." Heuristic names are capitalized, and you'll find the full set of them collected in Appendix 2, *Test Heuristics Cheat Sheet*, on page 151.

Online Resources

As with all of the Pragmatic Bookshelf titles, this book has a web page. If you visit http://pragprog.com/book/ehxta/explore-it, you can participate in discussions with other readers and with me. You can also report errors (and if you find any I certainly hope that you will take a moment to tell me about them).

Happy exploring!

Elisabeth Hendrickson

elisabeth@testobsessed.com

February 2013

Part I

Establishing Foundations

CHAPTER 1

On Testing and Exploration

No matter what your job title, you most likely find yourself testing on a regular basis. Testing is an integral part of creating anything. Until you test—interact with the software or system, observe its actual behavior, and compare that to your expectations—everything you think you know about it is mere speculation.

In *Portraits in Silicon [Sla89]*, Robert Slater tells the story of the team that built one of the earliest computers, the ENIAC. Early computers were massive, filling entire rooms. If you were to examine the insides of one of these machines, you would see racks of components with bundles of wires running between them. Thus the choice of wire became a crucial design decision. As Slater explains:

> Then there was the potential problem of mice eating the wire. As a test some mice were put in a cage, starved for some time, and then offered different kinds of wire. When it was discovered that the mice loved the type of wiring that had been planned for ENIAC, other wiring, which passed the mouse test, was chosen.

Notice that the team members recognized a risk and turned that into a question they could answer. They did not speculate about rodent dietary habits but instead offered up various types of wire to hungry mice. They used the results of their experiment to inform their actions. This is the essence of testing: designing an experiment to gather empirical evidence to answer a question about a risk.

Different types of testing answer different types of questions. If you want to know how well a system holds up under peak load, you might run a performance test. If you want to know whether a small bit of code does what the programmer intended, you might isolate that bit of code in a set of unit tests. If you want to know whether users will be able to find their way through the software without help, you might conduct a usability test.

This chapter explains what sets exploratory testing apart from other kinds of testing and how it fits in an overall test strategy.

1.1 Two Sides of Testing

It's been twenty years, but I remember the conversation as though it were yesterday. One of my coworkers, Marchell, pointed at an inch-thick stack of paper on her desk: test cases covering just a small fraction of the capabilities in the software package we were testing.

"It is so frustrating," she sighed. "No matter how many tests we write, no matter how many cases we execute, we always find the most serious bugs when we go off the script."

At the time, I didn't know the term *exploratory testing*, even though Cem Kaner had already coined it in his 1988 book, *Testing Computer Software [Kan88]*. I just knew that Marchell was right. No matter how many test cases we added to our test suite, we still found surprises when we deviated from the scripts.

In the two decades since that conversation, I have seen that pattern repeated many times: no matter how many preplanned test cases the team executes, there are still surprises to be found.

When an organization releases software into the wild, the surprises can be even worse.

Users do the craziest things. Production data has a nasty tendency to look different from concocted examples. Actual configurations aren't as neat and clean and controlled as test machines.

The real world is a messy place.

It's frustrating but undeniable: you simply cannot plan tests in advance to cover every condition. There are too many variations in data, configurations, interactions, sequences, and timing. If you try to create a comprehensive set of tests to cover every possibility, you'll spend all your time writing tests and have no time left over to execute them.

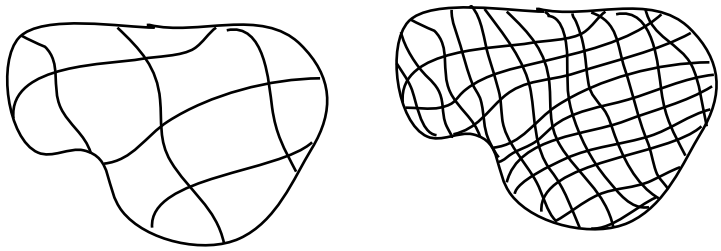
What you need isn't the perfect set of comprehensive test cases. Instead, you need a test strategy that answers two core questions:

1. Does the software behave as intended under the conditions it's supposed to be able to handle?
2. Are there any other risks?

Checking

You can answer the first question with tests that you design in advance to check that the implementation behaves as intended under supported configurations and conditions.

You can visualize these checks as a net of tripwires that are triggered whenever the software's behavior violates expectations, as shown below. The better the coverage offered by the checks, the finer the weave in the net.



However, even if you've woven a fine net, you still need to answer the second question. This is where exploration comes in.

Exploring

Exploratory testing involves scouting around the areas that the net doesn't cover. You interact with the implementation, designing and executing tiny experiments in rapid succession using the results from the last experiment to inform the next.

As you discover potential risks, you probe deeper. You use your ability to observe and analyze to adapt your investigation on the fly. Your experiments give you empirical evidence about the capabilities and limitations of your software. Along the way, you uncover new questions needing answers and you plan for additional types of tests.

Exploration offers a way of navigating through the infinite possible variations to steer toward the risks in a way that your preplanned tests cannot. To discover additional surprises, repeatability will not help you—variation will.

However, the two questions represent two facets of testing: checking that software meets expectations and exploring for risk. Neither checking nor exploring is sufficient on its own.

Tested = Checked + Explored

You're not done testing until you've checked that the software meets expectations and you've explored whether there are additional risks. A comprehensive test strategy incorporates both approaches.

As you read through the rest of this book, keep in mind that it addresses only the exploration side of the equation. This is a guide to techniques for discovering surprises, not a comprehensive discussion of all aspects of software testing.

1.2 Essential Elements of Exploratory Testing

One of the most widely quoted definitions of exploratory testing comes from James Bach's 2003 paper, "Exploratory Testing Explained." James said, "Exploratory testing is simultaneous learning, test design, and test execution."

This style of testing requires that your brain be fully engaged at all times. This is evident in Cem Kaner's definition of the term:

> Exploratory software testing is a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the value of her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project.[1]

I use a variation on James's definition to explain the practice, adding one more piece. The definition of exploratory testing I use is this:

> Simultaneously designing and executing tests to learn about the system, using your insights from the last experiment to inform the next

Each part of that definition is important: designing tests, executing tests, learning, and steering. Let's take a look at each of these aspects in detail.

Designing Tests

Test design involves identifying interesting things to vary and interesting ways in which to vary them. There is already a wealth of literature on the topic, including classics like Glenford Myers's *The Art of Software Testing [Mye79]* and Boris Beizer's *Software Testing Techniques [Bei90]* as well as the more recent and comprehensive overview by Lee Copeland, *A Practitioner's Guide to Software Test Design [Cop04]*. These books cover techniques like boundary value analysis, decision tables, and cause-effect graphing, as well as deriving

1. http://kaner.com/?p=46

tests from design models such as state diagrams, sequence diagrams, and flow charts.

All these test design techniques are still relevant in exploring. The more familiar you are with test design, the better you are able to design good experiments on the fly.

Executing

When exploring, you execute as soon as you think of a test. This is one of the key attributes that distinguishes exploratory testing from scripted testing. This immediacy of the execution distinguishes exploration from other testing approaches. You don't design your tests all in advance before you start executing them. You start executing right away. This is crucial: until you execute your test, you don't know what follow-up questions will occur to you to investigate. Immediate execution enables you to steer your investigation toward the most interesting information.

Learning

As you explore, you discover how the software operates. You learn about its quirks and peculiarities. You watch carefully, looking for subtle clues about where there might be a nest of bugs lurking. Observing is crucial: the better you are at observing, the more you'll learn. It's also harder than it sounds. You have to look past what you expect or hope to see in order to see what's really happening. (Chapter 3, *Observe the Details*, on page 25, provides advice on developing your observation skills.)

Steering

With each experiment you execute, you gain a little more insight into how the software behaves. You notice what kinds of conditions the software does not handle well and use that knowledge to push even harder. You use your curiosity, fueled by what you've learned so far, to suggest the next most interesting piece of information to uncover. Steering while focusing on the most important information to discover is one of the core skills of a master explorer.

1.3 Working in Time-Boxed Sessions

Exploring can be a completely open-ended endeavor. Without some mechanism for structuring and organizing your effort, you could spend hours or days meandering aimlessly through the software and wind up with no interesting or useful information to share.

In answer, Jon Bach and James Bach came up with the practice of session-based test management (SBTM).[2] In it, you structure your time into a series of time-boxed sessions. You establish a focus for your session in advance. (Chapter 2, *Charter Your Explorations*, on page 11, discusses how to establish and capture that focus.) During the session you explore fluidly, designing and executing tests, moving from one experiment to the next, without pause.

While in each session, you take notes so you know what you explored and what information you found. However, your notes are for your use. You'll refer to them when you debrief with stakeholders, but they are not like traditional test cases or test reports. Your raw notes will do others very little good. You might make notes about test ideas, questions, risks, surprises, additional areas you want to explore, and bugs.

At the end of the session you capture the information you need to convey to others. You might capture your observations about the capabilities and limitations of the area you explored in writing, or you might sit with your stakeholders to tell them what you discovered in person. If you found bugs that need to be reported, you report the bugs. If you have questions, you seek out someone who can answer them. The sessions provide periodic stopping points for you to distill your findings and consider the best area to explore next.

1.4 Practice Sessions

A key theme in this chapter is the difference between checking and exploring and the notion that a comprehensive test strategy requires both.

Take a moment to reflect on your current test strategy. Start by writing a list of questions that you expect testing activities to answer. For example, you might have very general questions, like these:

> Can a user actually use the software for its intended purpose? Do the basic workflows work?

You might also have specific questions about capabilities or interactions, such as this:

> How does the discounting feature interact with the bundling feature?

You might have questions about overarching concerns or characteristics:

> Will the software fail gracefully if it's overloaded?

Brainstorm a list of questions. When you run out of steam, review your list. Consider how the current test strategy answers each question. Ask yourself

2. http://www.satisfice.com/articles/sbtm.pdf

if each question is better answered with predesigned test cases (checks), through exploration, or through a combination of the two.

If you find that you would only use one aspect of testing—either checking or exploring—for every single question on your list, come up with questions that can only be answered with the other kind of testing.

For example, if all your questions lend themselves to exploration, consider questions related to basic capabilities that should always work.

On the other hand, if all your questions lend themselves to preplanned checks, consider categories of risks that you would only be able to find through active investigation: unintended consequences and side effects, surprising interactions, or unanticipated usage scenarios. (If you are still having a hard time imagining situations where exploration is applicable, you might want to finish reading the rest of the chapters in part 1 and then return to this exercise.)

CHAPTER 2

Charter Your Explorations

For just a moment, consider a different kind of exploration: exploring territory. Think of Columbus sailing west on a mission to find a new route to the Far East. Or imagine adventurers like Edmund Hillary, who climbed Mount Everest, or Robert Edwin Peary, who led one of the first expeditions to the North Pole.

Exploring software has much in common with exploring territory:

- Any number of surprises and adventures wait you (including bugs).
- You can use tools to support your journey, but the most important tool you carry is the one between your ears.
- Sometimes exploring is a fun romp, sometimes it's a slow slog, and sometimes you'll be treading on treacherous ground.
- If the map and the territory differ, believe the territory.

Further, in exploring both territory and software it's all too easy to hike off into the wilderness only to become thoroughly lost and go around in circles. However, that's not what great explorers do. They don't just set out with some vague notion that they'd like to see something new. They have a specific goal. They prepare for their adventures. They bring along help and supplies.

In this chapter you will discover how to make sure your explorations stay on course by using charters. You will learn the essential elements to include in a charter, as well as where charters come from. In subsequent chapters you'll learn how to use those charters as a guide.

2.1 Chartering the Expedition

In a letter dated June 20, 1803, Thomas Jefferson, the third president of the United States, gave the explorers Lewis and Clark a mission: discover a route

across the continent from a starting point near St. Louis, Missouri, out to the Pacific Ocean.

Jefferson's letter goes on for pages and in great detail. In it, he specified these points:

- *Where they were to explore*: the Missouri River and connecting waterways
- *The resources they would have*: equipment such as boats, tents, surveying tools, weapons, and even presents for the natives they would encounter
- *The information they were to seek*: trading routes. Jefferson explicitly directed Lewis and Clark to seek "direct and practicable water communication across this continent for the purpose of commerce."[1] He was not looking for nice picnic spots or locations for future national parks.

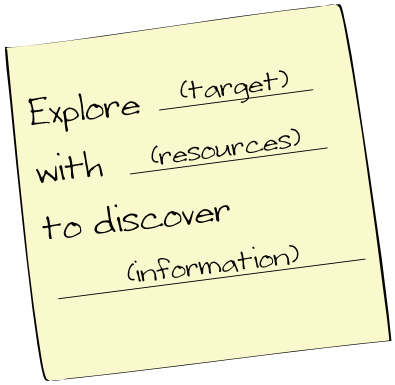
Three years and three months later, Lewis and Clark returned to St. Louis as national heroes. They had covered seven thousand miles and mapped the trail they followed through the interior of the North American continent toward the western sea. You can see the expedition route here:

Like Lewis and Clark, your ultimate goal in exploring is to discover information of interest and value to your stakeholders (the people on your team and any others who have a vested interest in the information you discover). The essential elements of the charter Jefferson gave his explorers suggests a simple template for charters, as discussed in the next section.

1. http://www.loc.gov/exhibits/jefferson/168.html

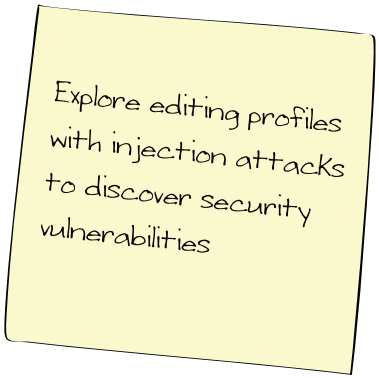
2.2 A Simple Charter Template

Although Jefferson's letter to Lewis and Clark went on for pages, the essential information it contained fell into the three broad categories above: where to explore, what resources were available, and what information to discover. This suggests a simple three-part template:



- *Target:* Where are you exploring? It could be a feature, a requirement, or a module.
- *Resources:* What resources will you bring with you? Resources can be anything: a tool, a data set, a technique, a configuration, or perhaps an interdependent feature.
- *Information:* What kind of information are you hoping to find? Are you characterizing the security, performance, reliability, capability, usability, or some other aspect of the system? Are you looking for consistency of design or violations of a standard?

Consider an example. Imagine you are working on software in which users can update their personal information in a public profile. You might create several charters related to this feature. If you were concerned about security, you might have a charter like this:



Note that this charter focuses specifically on security concerns. It's not about how well profiles display international characters or long names or any of the myriad other concerns someone might have about the feature. This charter is specifically about exploring using inputs that represent injection attacks.

While exploring against the charter, if the software had a web-based interface you might try a JavaScript attack like this:

```
<script>alert("HAH! Pwn3d!");</script>
```

You might try an SQL injection attack like this:

```
x'; DROP TABLE CUSTOMERS;
```

Alternatively, you might have a charter that calls for exploring the new feature of editing profiles with other existing features, such as authentication:

In executing this charter, you'll vary the mechanism used for authentication. Imagine that the software you're testing allows users to log in through an external service such as Twitter or Facebook or through its own login page.

Or perhaps the software supports a "Keep me logged in" feature. Any of these would be interesting variations to explore with editing profiles.

Notice the differences in the kinds of actions each charter suggests. One encourages you to channel your inner hacker; the other points more firmly toward variations in happy path scenarios. This level of variation is both expected and significant. Different charters invite different kinds of exploration. This is why you create charters: as a reminder to focus on a particular kind of information or risk while exploring.

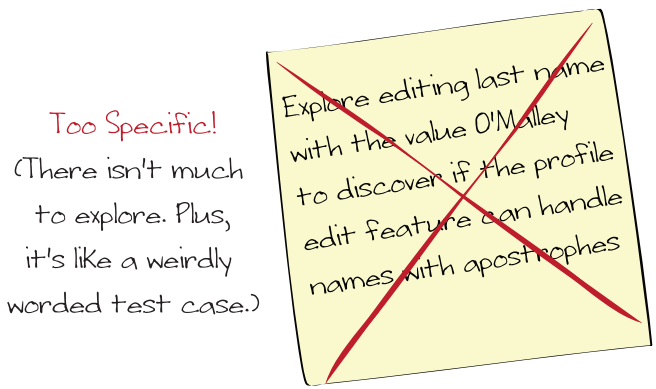
It's important to note that, like any other template, this charter template is just a guide. Templates are useful because they remind you of all the pieces that need to be present. A template can't fit every situation, however. So don't try to force-fit every charter into the template; you'll end up constraining your thinking by trying to shoehorn rich and complex ideas into a simplistic form.

When you're inexperienced with framing charters, use the template. As you gain experience, focus more on capturing the intent of a session than on filling in the blanks in the template. "Find ways that a customer might not be able to complete a valid purchase" is a perfectly fine charter. So is "Experiment with invalid values when updating user profiles."

This begs the question of what makes one charter better than another, which is the subject of the next section.

2.3 Good Charters

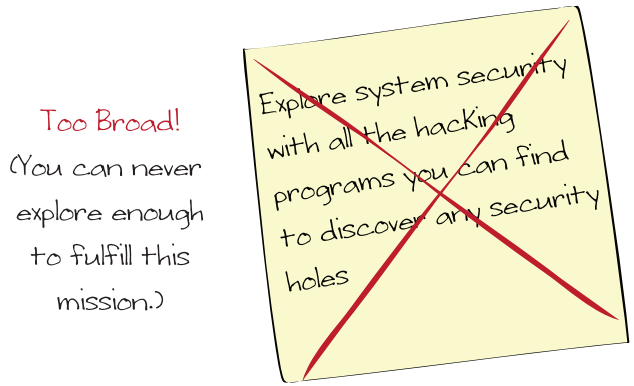
A good charter offers direction without overspecifying test actions. As an example, the following isn't a charter; it's a test case.


Too Specific!
(There isn't much to explore. Plus, it's like a weirdly worded test case.)
Explore editing last name with the value O'Malley to discover if the profile edit feature can handle names with apostrophes

When charters are too specific, they become just a different (and weird) way of expressing individual tests. We end up spending a lot of time on test documentation with very little benefit.

On the flip side, charters that are too broad run the risk of not providing enough focus. You won't know how to tell when you're done exploring if the target is too big.

For example, consider this charter:


Too Broad!
(You can never explore enough to fulfill this mission.)
Explore system security with all the hacking programs you can find to discover any security holes

It's so vague that you would never finish the mission. It calls for exploring the entire system with a large and undefined set of resources. You could spend weeks investigating it and still not be sure you discovered all the important risks and vulnerabilities.

Rather than a single huge charter, it is better to craft multiple charters, where each focuses on a single area and/or a specific type of security hole:

Explore input fields with JavaScript and SQL injection attacks to discover security vulnerabilities
Explore pages requiring login with spoofed URLs and POST parameters to discover if users can gain access to content they didn't purchase

A good charter is a prompt: it suggests sources of inspiration without dictating precise actions or outcomes.

2.4 Generating Charters

Your explorations yield information, but unless your stakeholders value that information and use it to move the project forward, you're wasting your time.

To ensure the information you find has value, you need to work closely with your stakeholders to identify and frame charters targeted at answering the most valuable questions about the software.

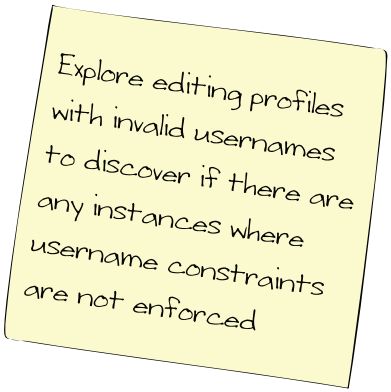
This section examines some of the sources that inspire your charters.

Requirements

Requirements discussions are an ideal time to start drafting charters. Let's see how this works by eavesdropping on a conversation between Alex, a tester; Pat, a programmer; and Binh, a business analyst. They're discussing the feature that allows users to update their profile information.

> ***PAT:*** *So, which of the profile fields should be modifiable?*
>
> ***BINH:*** *All of them.*
>
> ***ALEX:*** *Even the username? So someone with the username "fred728" can change his username to "iamfred" and then use the new username to log in?*
>
> ***BINH:*** *Yup!*
>
> ***ALEX:*** *Wow. I'm concerned about the possibility of violating the restrictions around usernames if we allow users to change their usernames after the fact.*
>
> ***PAT:*** *That won't be a problem. We've isolated all the username constraint logic to a single place in the code base. It will behave exactly the same as if a user is creating an account.*
>
> ***ALEX:*** *That sounds like something I should explore.*

Pat is asserting that there won't be any problems, but Alex has seen plenty of cases where the programmer was surprised by the software's behavior. He knows better than to assume Pat has the whole code base memorized. Alex makes a note:



The conversation continues. Alex and Pat have questions about the interactions between this feature and other capabilities that have already been implemented:

PAT: *Should users be able to update their profiles if their accounts are suspended?*

BINH: *Oooh. Good question. I'll have to think about that.*

ALEX: *I bet there will be interactions between updating profiles and account states. I'd better explore around that.*

Aha! Binh might come back with a simple yes or no answer to this specific question, but there is a deeper issue at play here. The first clue is that Binh says, "I'll have to think about that." The second clue is that there is a potential for an interaction between the new feature and existing capabilities.

Any time a question reveals uncertainty, ambiguity, or dependencies, there's something important to be explored during development. So Alex captures a charter:

As the discussion continues, Alex hits on an idea that wasn't on Binh's radar:

ALEX: *Users see their profile information on their account page and also on the dashboard. So far we're talking about editing their profiles from the account page. Should we also allow users to update their account information from the dashboard?*

BINH: *I think that's out of scope for now, but it's something we should consider in the next update. I'll add it to my list.*

Exploring can reveal opportunities to add new requirements as well as find risks or problems. When you begin questioning and exploring, you can watch for such opportunities, then review your suggestions with your stakeholders.

Implicit Expectations

In this example, Pat and Alex are asking Binh about his expectations. However, no matter how much Pat and Alex probe, Binh will still have additional expectations that he doesn't even think about expressing. Binh might think that a given expectation is too obvious to mention. Here's what one product manager said to me when I asked about the interaction between a new feature and the existing security features: "We have a security model in this system. New features have to honor that security model. I just expect you guys to take that into account without my saying it for every single feature."

Another example of implicit expectations might include crosscutting quality criteria such as reliability, scalability, or performance. If the functionality works as specified but the response time increases from under a second to over a minute, there's a problem, even if the response time wasn't explicitly stated as part of the requirement.

Whenever you recognize an implicit expectation that deserves exploration, capture it as a charter.

Charters Align Goals

The requirements discussion is an ideal time to get feedback on the extent to which your ideas about the most important risks match the rest of your stakeholders' concerns. As you think of possible charters, you can ask your stakeholders if they would value the information those charters might reveal. Pat, Alex, and Binh might consider questions like these:

- "Should we look for possible performance implications?"
- "If there were problems with legacy data, would we want to know?"
- "If we could find a way that users could put their accounts into an unusable state, no matter how crazy, we'd fix it, right?"

Asking these questions is important because you don't want to spend a lot of time discovering information that no one will ever take action on. It's a waste of time. For example, Binh might say, "If users do something really crazy to their accounts, the Help Desk can get them sorted out. Update Profile should use the same validations as Create Account, but nothing beyond that." This would imply that it's worth exploring updating the username, but it isn't worth spending days exploring conditions that go beyond the basic validation that Pat says is already built in.

Of course, you might disagree with your stakeholders about risk. The best time to surface that disagreement is during a discussion like this, before you

have spent hours exploring. Perhaps you see a particular kind of risk that the rest of your stakeholders don't see. Discussing the risk in advance can either allay your concerns or raise your stakeholders' awareness.

Stakeholder Questions

Questions surface throughout the development cycle. Indeed, sometimes the very best questions come up when mulling the implications of a given design decision or when mapping out a connected set of features. For example, if you were working on software that had both privacy settings and a messaging feature, your stakeholders might become concerned about their interaction:

> How do the existing privacy settings interact with our new messaging feature?

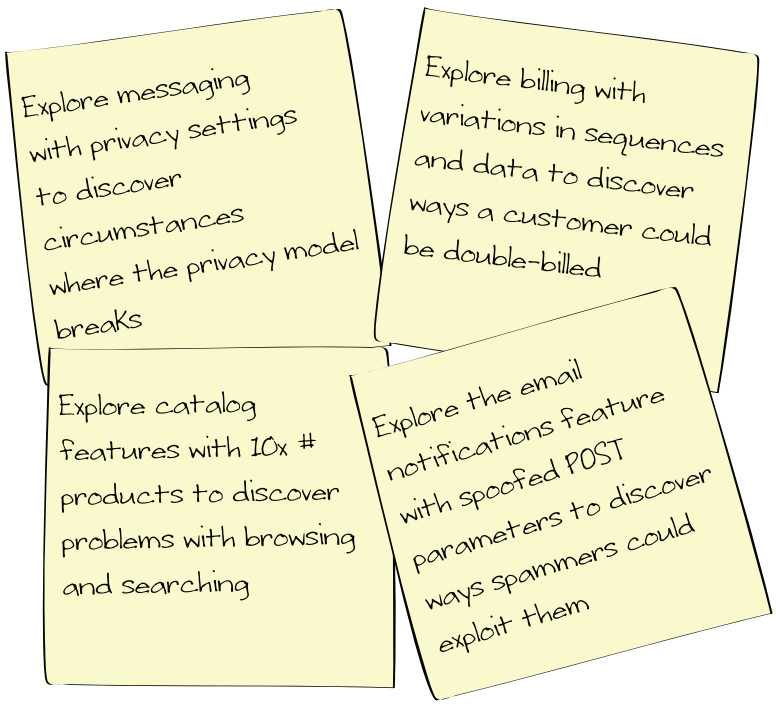
Your stakeholders might wonder what will happen in the future:

> What will happen if we add ten times the number of items to the catalog?

Your stakeholders might become worried about possible risks:

> Could a hacker hijack the email notification system to send spam?
>
> Are there any circumstances under which a customer could be double-billed?

These questions can form the basis for charters like these:


Explore messaging with privacy settings to discover circumstances where the privacy model breaks
Explore billing with variations in sequences and data to discover ways a customer could be double-billed
Explore catalog features with 10x # products to discover problems with browsing and searching
Explore the email notifications feature with spoofed POST parameters to discover ways spammers could exploit them

Existing Artifacts

The source code can even yield interesting charter ideas, particularly if you happen across a code comment like this one:

```
// I don't know why this works, but it does. Don't touch it.
```

Other existing artifacts associated with the software can yield chartering ideas. The bug database is likely to offer up a wealth of insight about historic areas of risk. Scanning the logs from support calls can give you insight into the risks that have historically bitten customers.

New Realizations and Discoveries

Chartering is an ongoing process. You start chartering as soon as anyone starts discussing requirements, and you continue identifying charters throughout development. As you explore, it's also normal for you to realize that the charters you have mapped out only barely scratched the surface.

You can tell this is happening when you start executing sessions against a charter only to discover that you're continually tempted to explore in directions that are decidedly off-charter and you're afraid to ignore these temptations for fear that you won't remember to come back.

Such temptations are a cue that you need to jot down additional charters to pursue in later sessions.

2.5 The Nightmare Headline Game

Another source of inspiration for charters can stem from our worst fears. A classic risk management technique involves imagining a catastrophic failure and then backing your way into discovering risks that could lead to that failure.

This particular activity makes for a great group exercise. It benefits greatly from the creativity generated in a group brainstorm with people who represent a diverse set of perspectives. In addition, it can help those who usually think in terms of creating, such as developers, business analysts, and product managers, to shift into thinking about what could go wrong and how to detect vulnerabilities.

To run this exercise, gather the participants in a room stocked with paper and markers.

Step 1: Set the Stage

Ask participants to picture themselves in their usual morning routine:

> You're getting ready for the day. You're feeling a little sleepy and moving a little slowly. You check the news. A screaming headline greets you. It's a major disaster. All the news outlets are covering it. Suddenly you're wide-awake, your adrenaline pumping, because the screaming front-page headline somehow involves the software you work on.

Pause for a moment to allow the participants to think. Then ask:

> Now, what's the headline?

Step 2: Gather Headlines

Ask participants to write down their answers on the paper provided. When everyone has something written down, ask participants to share their headlines.

Imagine you get headlines like these:

> *Shopping Cart Software Adds 58% Premium to Bill: Holiday Shoppers Outraged at the Overcharges*
>
> *Man Surprised by Shipment of 827 Garden Gnomes, says "But I Only Ordered One!"*
>
> *Billed, Not Delivered: Software Glitch Causes Consumers to Be Billed for Merchandise That Never Ships*
>
> *Security Hole in Shopping Cart Software Enables Hackers to Steal Credit Card Info*

Encourage the group to have fun with this and embellish the headlines with details.

Step 3: Choose a Big Risk to Work On

Ask the group to review the list of nightmare headlines to find a nightmare that stands out. You'll want to choose one that is plausible, related to the team's software, and interesting to discuss.

Ask the group to choose one headline to start with. If there isn't an immediate consensus, you can use multivoting (see *Using Multivoting*, on page 23) to discover what risk is most interesting to the group as a whole.

Step 4: Brainstorm Contributing Causes

Write the chosen headline at the top of a sheet of flip chart paper. Then ask the group this:

> What could possibly cause this problem?

Ask participants to write down possible causes on large index cards. For example, let's imagine we decide to work on this headline:

> *Man Surprised by Shipment of 827 Garden Gnomes, says "But I Only Ordered One!"*

Using Multivoting

Multivoting is a facilitation technique for prioritizing a list of items.

With this technique, all participants get multiple votes to distribute among the choices in whatever way they see fit. They can split their votes among the choices or put all their votes on a single choice.

Here's how to conduct multivoting:

1. Post the list of choices on a large piece of paper, like a flip chart pad.
2. Before you start the voting, ask if anyone needs any clarification about what the choices mean. It's important to make sure everyone knows what's going on.
3. Each person gets three votes. (You can give everyone dot stickers to track votes or simply make pens available to participants so they can mark their choices on the list.)
4. Ask participants to indicate their votes on the list. This can take a few moments if there are a lot of participants, so it can be a good idea to combine multivoting with a short break.

Contributing factors for this imagined disaster might include these things:

- Invalid inputs
- Hitting Refresh or Back in the browser window
- Clicking the submit order button multiple times
- Integration issues between the web server and the warehouse system

Step 5: Refine Causes into Charters

Post the list of contributing factors where everyone can see them.

Ask participants to work in groups of two or three. Have them choose any of the contributing factors and then identify charters for exploration that might reveal the problem. For example, charters inspired by the contributing factors in this example could include such ideas such as these:

> Explore modifying quantities with variations in input data to discover ways to inadvertently order more than intended.
>
> Explore the purchasing flow with browser operations such as Refresh, Back, Bookmark, and Form Submit to discover ways to inadvertently order more than intended.
>
> Explore the integration between the shopping cart and the warehouse system with variations in quantities and items to discover ways in which fulfillment could go wrong.

2.6 Planning Charters

If you are accustomed to planning large-scale test efforts in advance, it may be tempting to plan out all your charters before you begin executing against them.

The problem is that you cannot know until you start exploring what kind of information you will find. If you plan all your charters in advance, you will be less able to adjust your exploration to take that new information into account.

You might identify some of the most important charters in advance. However, chartering is not the same as traditional test planning. It is a much more fluid process. You are constantly revisiting your charters throughout the entire development life cycle to make sure you're always focused on discovering the information your stakeholders care most about. To put it another way, take your exploration one mission at a time.

2.7 Practice Sessions

Charters focus your exploration on the most important information to discover. When you frame your charters, you target a specific area to explore, identify resources or dependent features you want to explore with, and are clear about the kind of information you intend to find.

In order to identify the most useful charters, you need to ask plenty of questions, consider potential interactions and associated vulnerabilities, solicit ideas about risks, and listen carefully for unanswered stakeholder questions.

Practice your chartering skills. Try drafting charters for exploring a software package you're currently working on. Remember to choose specific areas to explore, and then draft charters related to these areas:

- Stakeholder questions
- Interactions between features
- Imagined nightmare scenarios
- Test ideas you had during a session that would have led you off-tangent for your current charter

In the next several chapters we'll focus on the work involved in actually exploring in sessions. We'll also revisit chartering in Part II, *Adding Dimensions*, on page 57.

CHAPTER 3

Observe the Details

Testing software might seem like a straightforward, linear process. Input a value. See a response. Check the actual response against the expected response. If they match, the test passes. If they don't, the test fails.

It's not that simple. Just because you see something that you expected to see does not mean that all is well with the system. There can be serious problems lurking just out of view. Software is like an iceberg: much of the action is below the water line.

That's why one of the most important skills good explorers possess is the ability to observe. This sounds simplistic, like something anyone should be able to do without even thinking. It turns out that observing is not all that simple. It's all too easy to miss critical information.

Good observation necessitates looking past what you expect or hope to find in order to see what's really there. You can't accept anything at face value; you have to peer behind the curtain. You watch behind the scenes using consoles, logs, or monitoring applications. You use private interfaces to control the application. You stay alert for any hint of surprising behavior.

Far from being simplistic, learning to observe is among the most difficult things for an explorer to master. This chapter examines concrete ways to improve your powers of observation.

3.1 But Did You See the Moonwalking Bear?

There is a well-known series of videos produced by Transport for London about cycling safety. One of them is colloquially known as the moonwalking bear ad.

The ad begins with a voiceover:

"This is an awareness test. How many passes does the team in white make?"

As the video continues, you see two teams of four people each. One team is in white shirts and the other in black. The players on each team dart around a basketball court, intermingling, passing the ball among their fellow team members.

The first time I saw the video, the perfectionist side of my personality took over, and I obsessed about getting the count right. It was difficult to track the movement of just the white team's ball and ignore the black's: the players and the two basketballs moved so quickly. I concentrated hard, intent on watching the little orange globe moving between the white-shirted players. Despite my intense focus, I was not confident in my answer at the conclusion of the video.

The video continued: "The correct answer is 13." Dang it. I'd overcounted by one. How did that happen?

Before I could figure out where I had gone wrong, the voiceover interrupted my thoughts: "But did you see the moonwalking bear?"

What? What bear? You're kidding me. This is a joke. There was no bear.

The video replayed. As I watched the video from a different mental perspective, I could see the bear. It was so obvious it was ridiculous. A dude in a bear suit sashayed through the players as they passed the ball.

Still convinced this was a trick, I watched the video again all the way from the beginning. Sure enough, there was the bear. It had been there the whole time and I had been completely unaware of it. The dark fur suit blended in with the black team. Try it for yourself. Search on YouTube for "moonwalking bear" to find the video.

Here's the tagline for these cycling safety videos:

> It's easy to miss something you're not looking for.

Indeed, it is. That's why this chapter is all about learning to see the dancing bear in your software.

The Transport for London videos highlight the phenomenon of *inattentional blindness*: an effect where paying attention to one thing can result in not seeing something else significant, even if it is in plain sight.

Cem Kaner and James Bach deserve the credit for highlighting the phenomenon and its related research, particularly the work of Daniel Simons, a professor in the department of psychology at the University of Illinois at

Urbana-Champaign. Simons created the original video that was a precursor to the Transport for London ad. In it, he had basketball players passing around a ball, but instead of a moonwalking bear, he showed a gorilla walking across the field of view.

In an article about inattentional blindness on Scholarpedia.org, Simons writes this:[1]

> The more people focus on aspects of their visual world other than the detection of unexpected objects, the less likely they are to detect such objects.

When you consider that testing necessitates careful observation, it's apparent how relevant this insight is for anyone who tests software. It implies that the more you concentrate on only a single dimension of the software, the more likely you are to miss critical surprises along other dimensions.

What this means to you as an explorer is that in order to increase the odds of seeing surprises, you have to make a point of changing your perspective. Direct your attention to different aspects of the software. Watch different parts of the screen. Expect surprises and look for them intentionally.

3.2 Digging Deeper

Back when most software was distributed on CDs, I was a test manager on a consumer application that ran on Windows machines. Part of the project involved creating and testing the installation program. With just a few weeks to go before we burned the golden master CD, we had found no bugs in the installer, so we thought it was good to go.

We gave a preliminary copy of the CD to our tech support group so they could prepare for the launch. Shortly thereafter, I got a phone call from the head of support. "Hey," he said. "We installed the software, but it doesn't work!"

I couldn't believe it. We were just about to ship. We'd been testing for months. Of course it worked. I assumed the problem must be human error. I went over to the support area to investigate. Sure enough, the software was installed, but it wouldn't run. When I dug a little deeper, I discovered why: key files were missing.

How did we get so far into the process without discovering fatal flaws in the installer? I talked with the installation tester. Because we had to test the installer on so many configurations, he had been a little rushed. So he just checked whether or not the installation wizard prompted the user for the

1. http://www.scholarpedia.org/article/Inattentional_Blindness

correct configuration choices and then completed without error. He had not verified that the program was actually installed correctly or that it would run.

The installation tester and I quickly remedied the situation. We worked together to develop a checklist of actions to take post-install to determine whether or not the program had been installed correctly. Items on the checklist included these steps:

- Make sure that all the files are installed in the right place on the hard drive.
- Verify that the registry values are added correctly.
- Launch the application and perform a handful of basic operations.

We then retested the installer in the various supported and unsupported configurations. Now that we had a more reliable method of determining whether or not a given installation was successful, we gained much more information from our tests and found a few more problems that would have prompted technical support calls if we had shipped with them.

Asking the Deeper Question

During initial testing, the installation tester was only asking a very superficial question: "Did the installation wizard complete without error?" By only focusing on errors visible in the GUI, he missed serious defects. The deeper question was, "Can I use the software I just installed?"

When testing, you have to look beyond the obvious and superficial questions. If you are testing reports, it's not enough to see that the reports ran. You need to check that they provide accurate information for a variety of interesting scenarios. If you are testing login, you need to do more than log in and see a "Welcome!" message. You need to attempt to access content to which you should, and should not, have access.

For any kind of feature you are testing, you can stop at the superficial questions or you can dig deeper. To get more—as well as more valuable—information, asking deeper questions yields better information.

Be Alert for Subtle Clues

Sometimes the cues that tell us that something is amiss are even more subtle than an error in a system log. A tiny, barely noticeable flicker in a user interface can foreshadow a serious problem. An unexpected noise can alert us to unexpected activity. A rapidly increasing memory footprint can presage an out-of-memory failure.

When we're observing, we need to use all our senses. Consider:

- *Seeing*: Is the disk light flashing when you don't expect it to? Do you see unexpected windows appearing? Is there activity in the background that you don't expect? If you are testing a user interface, you might also take screen shots at key moments. You can then view the screenshots, flipping rapidly between them. Your eyes will be drawn to differences on the screen.
- *Hearing*: Do you hear the soft whirr of a hard drive spinning up when it is not supposed to?
- *Touching*: Is the machine running hot when the system isn't supposed to be doing anything particularly computing intensive?

If you are exploring software that's particularly prone to subtle issues, you might consider using screen-recording software to capture a session. You can then watch your session again to see clues you missed the first time around.

An Unexpected Noise

I had been testing the low-level system utility for too long. My eyes were glazed over from looking at the same screens for hours on end. Worse, I'd had very little sleep. The project had become a death march. I could no longer be sure that I would recognize a problem even if I saw it.

And yet I was still testing at 1 a.m., working at home. My family was asleep. I'd turned off the television. The room was dead silent. I hunched over the laptop, concentrating on verifying the most recently fixed bugs in preparation for a big release the next day.

Although I was not as alert as I should have been, after a while I became aware of a quiet "whrrr" noise breaking the silence every few moments. I paused what I was doing and just listened. There it was again. "Whrrr." Then silence.

I put my hand on the side of the laptop. Sure enough, a moment later I both heard the "whrrr" noise and felt a tiny vibration. The sound was coming from the machine: the hard drive was spinning up.

Intrigued, I turned off our software, shutting it down completely and killing off all of its processes. Then I waited. No "whrrr."

I turned our software back on and waited. Another "whrrr."

The subtle clue had led me to a crucial discovery: our software was causing the hard drive to spin up every few moments.

More experimentation revealed that the hard drive spinning up meant that our software would prevent laptops from going to sleep. That would mean drained batteries and cranky users. This would violate one of our core requirements. As a low-level system utility that ran in the background, our system could never interfere with the user's experience.

We delayed the release by a day so we could fix the problem.

I was lucky. I didn't go looking for that bug; it found me. If I hadn't been testing in a dead silent house in the middle of the night, I never would have noticed the problem.

3.3 Testability and Making the Invisible Visible

Lesson 137 in Cem Kaner, James Bach, and Bret Pettichord's *Lessons Learned in Software Testing [KBP02]* says, "Testability is visibility and control." So a testable system provides both visibility into its inner workings and ample control points to manipulate it.

The challenge is to determine how to monitor and control a system that was not designed with monitoring and control in mind. There are many options for you to try, such as these:

- You can run operating system monitors such as Activity Monitor on the Mac or Process Monitor in Windows to tell you how the program is using system resources such as CPU and memory and even the libraries the program depends on.
- You can use programs (or write your own little script) to monitor file system activity: creating, changing, or deleting files on a file system.
- You can watch the network traffic.
- If the system is web-based, you can use plugins like Firefox's Firebug to show you the requests and responses from the server.
- If the system involves a relational database system on the back end, you can create tables in the database specifically for writing test-related data. Then you can insert monitors: triggers on key tables that will alert you when values in those tables are inserted, updated, or deleted.

In short, although the software we are testing may not be particularly testable, external dependencies such as the file system, network, relational database, and operating system can all be monitored and to some extent controlled. That means we can make at least some of the inner workings of the system visible so we can observe more closely.

Of course, if you have the ability to add testability hooks into the software, that's even better. You could add a verbose logging mode so that the software reports on every action taken, including any relevant information, such as internal state. You could add hidden fields to a GUI that contain system information that users don't need but that makes it easier to tell what the software is doing at any given moment in time. You could even create a monitoring/diagnostics utility that shows relevant information while the software is in operation. (Just make sure that if you add testability hooks you either make sure they're safe to go into production or you take them out before shipping the software.)

3.4 Consoles and Logs

Consoles and logs are particularly valuable resources for information about the inner workings of the system. They can act as an early warning system, alerting you to error conditions that have not yet shown up in more obvious ways. They can also give you insight into the functioning of an application, since they typically reflect the most important activities in the system. Finally, they can help you discover the structure of the application even if you don't know what all the moving parts are.

For example, when I am testing a web application, I keep an eye on the JavaScript console, the web server log, and the application log. I am watching for anything surprising or out of place. When I am working on a UNIX-based system (or a Windows system with Cygwin installed so it has UNIX commands available), I can use the command `tail` with the `-f` option to stream log entries while I am exploring by typing a command like this at the command line:

```
tail -f production.log
```

If the log is too verbose, giving me too much noise and too little useful insight, I filter the stream. The UNIX command `grep` makes this easy. I can send the output from the `tail` command to `grep`. Imagine I only wanted to see log messages containing the word "ERROR"; I would use this command:

```
tail -f production.log | grep ERROR
```

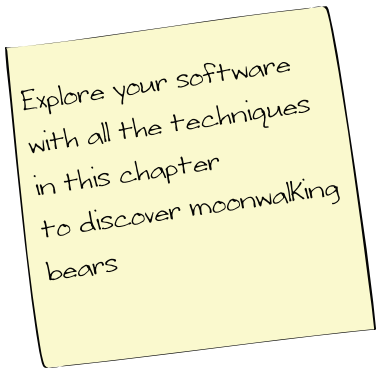
The log can also help you connect actions through the user interface with lower-level system calls. If you watch the log while performing various actions like creating new records, you'll discover what's happening under the covers. You may find that the system is doing more than you expected with apparently simple actions.

Here are a few lines from the production log for a Ruby on Rails site that I built. These log entries are the result of doing a search on the site.

```
...
Processing by ProfilesController#search as JS
Parameters: {"utf8"=>"\342\234\223", "search"=>{"direct_connections"=>"0",
  "others"=>"1", "search_location"=>"", "indirect_connections"=>"0",
  "search_text"=>""}}
[paperclip] Saving attachments.
[paperclip] Saving attachments.
...
```

Notice the word *paperclip* in the snippet. Paperclip is a Ruby gem for processing images. This particular website uses it for users' profile pictures. The surprise is that Paperclip is saving attachments whenever we search for users. That shouldn't be happening. Users don't see any bad behavior, but it's still a problem: eventually the site is likely to have performance issues resulting from the extra file saves on searching.

3.5 Practice Sessions



The first step in learning to observe more effectively is to recognize the opportunities to look past the obvious. Here are some things to try:

- Tune out the things you usually watch carefully; pay attention to details you usually ignore.
- Find a way to dig deeper. Consider an area of an application that you might explore, and ask yourself how you can get beyond the superficial observations to something that's more difficult to discover but more interesting and meaningful if you do.
- Discover at least one new monitoring application, console, or log file that you can watch while exploring.

Each of these activities enables us to see our software in a new light. Pay attention to surprises even if they aren't bugs. Surprises are an indication that we've successfully seen something new and uncovered new information.

CHAPTER 4

Find Interesting Variations

Once upon a time, I was interviewing a tester. I showed him a screen mockup and asked him what kinds of things he would want to test. He listed four or five scenarios he'd try. Then he leaned back in his chair and nodded to himself. "Yup," he said, "That'd do it."

He didn't get the job.

It's not that the four or five things he listed were somehow lacking. They weren't. He had good ideas. The problem was that he was completely satisfied with his answer. He didn't ask questions. He didn't even leave open the possibility that there was something he hadn't considered. He signaled clearly that he was done.

Even in a relatively simple system, there are an infinite number of variations to explore, including user interactions, sequences, timing, data, configurations, and environmental factors like network traffic or CPU load. That's why there's an old joke that "exhaustive testing" is when the tester is too exhausted to continue.

Given the infinite landscape to explore, being completely satisfied with a bare handful of experiments is downright dangerous. If you explore only superficially and then report that all's well, as I feared this interview candidate might, your stakeholders will make bad decisions, not realizing they are missing a big piece of the picture.

Good explorers recognize that there are an infinite number of variations and they cannot possibly cover all of them. Great explorers know how to hone in on the most interesting variations. There is a skill in analyzing a system to discover novel and yet realistic ways to manipulate it.

This chapter is about recognizing things to vary. Actually, because any testing effort involves varying how you exercise the software, much of this book is

about discovering interesting variations. However, this chapter in particular focuses on helping you learn to see those variations. You'll learn to see past the superficial, obvious things to find subtle or hidden variations that are overlooked all too often. You'll find ways to structure your search for interesting variations and to connect those variations with the techniques in later chapters. Above all, you will expand your thinking, prompting you to consider new dimensions for exploring your software.

4.1 Variables Are Things That Vary

This chapter is about variations, and a variable is the indivisible atomic unit of variation. Because the word *variable* comes with some baggage, this might require some explanation. If you have ever written a line of code, you might think of a variable as a named location in memory that you declare with statements like this:

```
int foo;
```

That's great knowledge; it's what a variable looks like inside a program. However, when you're exploring software, you're manipulating a different kind of variable.

> In testing, a variable is anything that you can change or cause to be changed indirectly while operating the software.

In particular, you want to find the variables that might affect the system behavior in interesting ways. There are three different kinds of variables to consider: the obvious ones that jump out at you, the subtle ones that are easy to miss, and the ones that are only indirectly accessible.

Obvious Variables

Some variables are immediately obvious. For example, if you are exploring a GUI that has fields on a form, the fields are variables because you can change their values. If you're exploring an API, the values you pass into the API calls are variables. Although such obvious variables are important, they're often the least interesting because they've typically already been tested extensively.

Although simply changing the values might not be that interesting, you can certainly choose interesting values. Different values exhibit different characteristics.

For example, consider software that takes in a binary input (a sequence of 0s and 1s) and converts it to a decimal value. If you input the value "101" or "011," you might expect it wouldn't make much difference. However, "101" is symmetric; "011" is not. If you choose an input value that is symmetric,

you won't notice if the program is reading the bits in the wrong order. By contrast, you'll most likely notice if the program interprets the binary number "011" as equal to the decimal value 6 instead of the decimal value 3.

Thus you see that variables are fractal in nature. Having found one thing to vary, such as the value of an input, you discover that you can also vary aspects of the values you choose, such as the symmetry.

Subtle Variables

The symmetry of an input is an example of a very subtle variable. Other subtle variables may be visible, but they are not intended to be changed directly, like parameters in a URL in the address bar for a browser. For example, imagine you're exploring a website and notice that the browser shows the address of the current page as this:

```
http://example.com?page=3&amp;user=fred
```

There are two variables after the ? character in the URL: the page number and the username. They're contained within the key/value pairs in the URL string.

Theoretically you're not supposed to manipulate such variables directly; they're set by the web application when you click a link. However, people do mess with URLs. Sometimes users change the URL for a good reason: they want to hop directly to a different page in a report or change a query without having to navigate through the UI. Or perhaps they have a stale bookmark. Other times malicious users are trying to find a security vulnerability.

So what happens if a page or user doesn't exist in the system? In some systems users see an ugly error.

Or what if they hack the URL so that it contains something like "user=foo;'drop table customers;"? In an insecure system, users manipulating the parameters passed to the server can cause very real damage. So exploring for these kinds of vulnerabilities is a good idea, especially when you find variables that weren't intended to be changed by users.

Another example of settings that users aren't supposed to change directly can include hidden preferences settings. Sometimes there are settings in cookies or configuration files that are supposed to be set only by the software. If users change these values, they may gain access to something they shouldn't. So as you're exploring to discover variables, look for hidden settings as well.

Indirectly Accessible Variables

Some of the most interesting variables are buried deep. They're things that can only be controlled indirectly, like the number of users logged in at any given time, the number of results returned by a search, or the occurrence or absence of a condition.

These variables are easy to miss, but if you can find them and exploit them, they often reveal critically important information about the system, as you'll see in the next section.

4.2 Subtle Variables, Big Disasters

Let's consider some famous software failures to see how important, and how difficult, it can be to identify subtle variables.

The Therac-25 Case

In the late 1980s, a number of cancer patients treated with the Therac-25 radiation therapy machine sickened after their treatment. Some died. The cause of death was radiation poisoning.

Investigation revealed that under certain circumstances the machine delivered an overdose of radiation rather than the safe medical dosage that the doctor prescribed. The investigation team concluded that the root cause was not a single failure but rather a complex set of interacting variables. Although hardware safety deficiencies contributed to the malfunction, the software had serious defects worth examining.

As Nancy Leveson explains in her book *Safeware [Lev95]*, at least one of the incidents happened because the technician entered and then edited the treatment data very quickly, in less than eight seconds. That eight seconds also happened to be the time it took the magnetic locks to engage, creating a short but devastating window of vulnerability.

The speed of input is a very subtle variable. The way users interact with the system is always a variable, of course, but the speed of input, and the fact that there was an important threshold at the eight-second mark, is quite subtle.

Yet it's also a potentially common situation: power users can generally manipulate systems with astonishing fluency. Using keyboard shortcuts and with lightning fast typing speeds, they may be able to enter data so quickly that they get ahead of the system. In response, the system may exhibit interesting behavior, such as unexpected errors or, as in this case, catastrophic failure.

Further, Leveson found that every 256th time the setup routine ran, it bypassed an important safety check. The number of times you've started a system is always interesting. Frequently systems do things differently the first time you run them. In this case the system worked differently the 256th time, the 512th time, and so forth, that it was booted up. That difference in behavior meant the system was vulnerable to potentially fatal malfunctions.

The Ariane 5

In 1996, the Ariane 5 rocket exploded spectacularly during its first flight. Investigation revealed that the root of the problem was a conversion of a 64-bit floating-point number with a maximum value of 8,589,934,592 to a 16-bit signed integer value with a maximum value of 32768. That conversion caused an overflow error. Compounding the problem, the system interpreted the resulting error codes as data and attempted to act on the information. As a result, the rocket veered off course. When it detected the navigation failure, it self-destructed as designed.

The conversion problem stemmed from differences between the Ariane 5 rocket and its predecessor, the Ariane 4. The control software was originally developed for the Ariane 4. However, the Ariane 5 rocket was significantly faster than the Ariane 4, and the software simply could not handle the horizontal velocity the rocket sensors registered.

Velocity is the key variable in this case. For a rocket, it's kind of an obvious variable. The tricky bit, and the less obvious detail, was that there was such a difference in the horizontal velocity for the Ariane 4 and the Ariane 5. So the platform on which the software is running is another key variable. If you have ever done cross-browser or operating system compatibility testing, you know how critical the platform on which the software is running can be.

The Mars Rover

In 2004, NASA lost contact with the Mars rover Spirit. NASA soon discovered that Spirit had encountered a serious anomaly and was rebooting itself over and over again. For a while, it looked like the rover's mission was over before it really began.

An article in *Spaceflight Now* explains that the problem was caused by the number of files in flash memory.[1] The rover had started collecting data on its flight to Mars. Some operations created numerous small files. Over time, the table of files became huge. The system mirrored the flash memory contents

1. http://www.spaceflightnow.com/mars/mera/040201spirit.html

in RAM, and there was half as much RAM as flash memory. Eventually the table of files swamped the RAM, causing the continuous reboots.

The key variable here isn't just the amount of disk space used; it's the number of files. Testing this by filling up the available space with a single giant file might not reveal the same behavior as having lots and lots of little files. So even when we talk about variables like space available on the disk, subtle details matter.

In exploring any system, there are variables and then there are variables within variables. It's variables all the way down. Becoming adept at taking note of interesting variables is the key skill that will enable you to explore absolutely any kind of software.

In the next section, you'll learn what to look for to spot subtle variables and how to manipulate them.

4.3 Identifying Variables

Until you become accustomed to looking for variables, it can be difficult to spot the subtle and inaccessible variables. The trick is to use common patterns of variables to help you learn to see the things you can vary in your software. This section offers a list of kinds of variables to watch for in your software.

Things You Can Count

Every system has things you can count. It might be a count of user accounts in the system or the number of times an account is logged in. It could be the number of phone numbers associated with a profile or the number of printers configured on a machine. Perhaps it's the number of files to be processed in a batch run or the number of records in a given file.

Counts are a subtle variable often overlooked until it's too late. Notice that two of the three stories above deal with counts: the Mars rover story hinged on having too many files, and one of the risk factors associated with the Therac-25 malfunctions was the number of times the setup routine had been run.

Once you find interesting things to count in your system, you can use these heuristics to vary the counts.

- *Zero, One, Many*: Do a search to return zero, one, and many records, and you may notice a common gaffe: messages that say "0 record found" or "1 records found."

- *Too Many*: Create conditions with more things than the software can handle, such as too many open connections or too many requests.
- *Too Few*: Create conditions with fewer things than the software expected, such as too few items to populate a list or too few line items on an invoice.

A count of zero is often interesting in its own right. Often software that is expecting a set of things does not handle the case where the set has no elements. Thus, zero is a heuristic in its own right.

For example, a desktop application I tested behaved badly if the computer had zero printers installed. For a more recent example, consider Chess Wars, an iPhone game designed to allow you to challenge your Facebook friends to a game of chess. When you tried to find someone to challenge, if zero of your Facebook friends had the Chess Wars app, your iPhone app exited unexpectedly.[2] When the application was first released, only the few people involved in creating it or beta testing it had it installed, so nearly all new users encountered this bug.

Relative Position

When you identify situations where things have a relative position, you can apply the Beginning, Middle, End heuristic.

For example I once tested a text editor. Pasting text at the very end of a line caused problems, but pasting text at the beginning or middle did not.

On another system that presented data in a list, deleting the very last item in the list always failed. You could delete the first item or items from anywhere in the middle of the list, but you could not delete that last entry.

You can also explore areas where you can vary position to discover if items take the correct position relative to the other items. You might have a stack of objects on top of one another, like elements in a drawing with a z-order determining which appear at the front or the back. You could vary the position of the elements from front to back to see if they are still rendered correctly. If you have a sortable list, you can create items that should sort to the top or the bottom of the list.

Be particularly alert for instances of items that are supposed to sort in numeric order but that end up being sorted alphabetically. In a numerically sorted list, 9 should come before 10. However, if the list is sorting alphabetically, 10 will come before 9. This is particularly a risk when you have data

2. http://getsatisfaction.com/blundermove/topics/application_closes

that is not, strictly speaking, a pure number. For example, in a list of IP addresses, you might expect it to sort 10.5.4.1, 10.5.4.2, 10.5.4.10. However, if the software is sorting alphabetically, this list will come out as 10.5.4.1, 10.5.4.10, 10.5.4.2.

It turns out that the Turkish *i* also provides an interesting case for positions and sorting. In the Turkish alphabet, there are two variations on the letter *i*, one with a dot and one without.

Files and Storage

Systems often make assumptions about where to find files or data. It might be the location of a file on the file system or the identifier for a hard drive volume. If you can change where a dependent resource lives, you've found a variable.

Changing up the locations can yield interesting surprises. If you move a resource in a distributed system to a location behind a firewall, you might find that the system can no longer access it. Installers often don't handle custom drive locations well. Or if the installer handles a custom location fine, the uninstaller might not. Any number of installers might fail when you attempt to install to any drive other than the default.

Geographic Locations

Software often has logic related to geographic locations: time zones, postal addresses and ZIP codes, elevations, and so on. If you vary addresses or geocodes, you may find that the system can only handle certain locations.

A decade ago, when mapping programs and routing were in their infancies, one routing algorithm worked fine as long as you specified two locations close together. One tester decided to try to plot the route from his work address to the city where he was born, which was about a thousand miles away. It was the first time anyone had attempted such a long route, and the system was completely unable to handle it.

In another case, users accessing a system in different time zones resulted in time stamps showing that a record was edited before it was created. Whenever you can change the physical location of a user or address, choose something far from the default location.

If you tend to use addresses like "123 Main Street" (or whatever equivalent is appropriate for your geographic area), you can increase the power of your exploration by introducing randomness into the location data you use. Use

Google Maps or MapCrunch to select a random location on the globe,[3] or use a test data generator service or tool like fakenamegenerator.com to generate random names and addresses.[4]

Formats

Any number of things have a defined format: dates, mailing addresses, file paths, URLs, the contents of a file, messages, and the list goes on. Any time something can look different and yet retain the same meaning, you have found a format that you can vary.

In some cases there are multiple valid formats:

- In the United States, phone numbers might be represented as either "(866) 867-5309" or "800-867-5309" or even "866.867.5309." They might have the international dialing code as well, like "+1 (866) 867-5309." In the United Kingdom, phone numbers can vary in length.
- Postal code formats vary by country. In the United States, postal codes are either a five-digit number or a nine-digit number represented with a hyphen between the first five and the last four numbers (such as 90051-0345). In Canada, postal codes are six characters and contain letters (such as "M4B 1B4").
- Dates are expressed as month/day/year in the United States (for example, "12/31/2012"), but in Europe they're day/month/year (for example, "31/12/2012.")
- Email addresses can look like "bob@example.com" or like "Bob Smith <bob@example.com>."
- IP addresses can be expressed in v4 format (like "127.0.0.1") or in v6 format (like "::1").
- Pictures might be in .png, .jpg, or .gif format. For that matter, you might try substituting .pdf or .eps files if your system handles images.

Invalid formats are also always fun. Violate domain-specific rules for specific kinds of data: a negative age, an IP address like "999.999.999.999," or a date of February 31.

If your system parses files, you can violate format expectations by corrupting the contents of the file. Throwing random garbage into the file is one way to do it. Another possibility is to create a file with nothing in it. Or you could

3. http://maps.google.com or http://mapcrunch.com, respectively.
4. http://www.fakenamegenerator.com

create a file that is mostly right but is missing some key piece. For example, if your system is parsing an XML file, omit a necessary node or make the XML invalid by removing a closing tag. All these variations are usually interesting.

Size

Obviously files have sizes. If your software imports files, try exploring it with empty or with massively huge files. Similarly, if your software manipulates data in a database, try it with a large database as well as with the more typical near-empty test databases.

Pictures have height and width, and sometimes systems scale or crop images to fit a predefined size. Explore with images of different dimensions as well as different file sizes.

Size matters with hardware configurations too. Installers often include logic to detect the size of a disk drive. Programs often check how much memory is available. One software package I tested misbehaved if the computer was configured with too much memory. Since it is difficult to imagine a system that has too much memory, this might be difficult to believe, but the software had a bug that only showed up when the computer was configured with extra memory.

Software often enforces boundary conditions for sizes around the powers of 2. For example, a common length limit for text fields is 256 characters (2^8). As you are exploring sizes, explore around the powers of 2.

Depth

Anything with a hierarchy has depth. XML data elements can be nested deeply. Files can live deep in the file system. Even equations can have a level of depth with nested parentheses. Exploring by spelunking varying levels of depth often yields interesting surprises and sometimes errors.

An HTML parser I once tested appeared to handle nested tables fine, but the time to parse the file increased exponentially with each level of nesting. The difference wasn't noticeable until I created a file with four levels of nesting—that took a minute or two to open. Encouraged, I tried five levels of nesting. Forty-five minutes later the program was still struggling to parse the file.

The version of Excel that I use has a bug in its floating-point calculations that only manifests with three or more levels of depth of parentheses. Consider the following equation:

```
=1-(100*(1-0.99))
```

It produces the correct result, 0, in my version of Excel. However, when I add one more level of parentheses I get a very different result. This equation looks like it should produce the same result as the equation above:

```
=(1-(100*(1-0.99)))
```

It does not. It yields the result -8.88178E-16.

Timing, Frequency, and Duration

Timing and user actions are always variables. The result of varying timing and sequences of actions can include timeouts, interruptions, and error conditions. Remember the Therac-25 case: at least one of the malfunctions resulted from the operator entering and editing the treatment plan in under eight seconds.

Frequency is an aspect of timing. Try doing actions often and repeatedly. For example, if you refresh Twitter too often you'll hit the requests ceiling and the Twitter server will return an error code. This is expected behavior, but if you have an application dependent on Twitter, it may not handle the error code correctly.

Duration is another aspect of timing. Try varying the duration of your actions, like keeping files or windows open for long periods. One of my favorite tests is to bring up an application and then leave it running overnight. For most apps there is no issue, but some encounter problems. One application I tested had a slow memory leak, so leaving it open overnight meant a crash notification was waiting for me in the morning. Another application I tested had a timeout that terminated sessions after a period of inactivity. Leaving unsaved changes on the screen for ten minutes or more resulted in the loss of whatever changes you were making.

You can use state models to find additional ways to vary timing. Chapter 8, *Discover States and Transitions*, on page 75, discusses how to analyze states, events, and transitions in detail.

Input and Navigation

Even just the way you enter data or manipulate the software can be a variable. If you are exploring a GUI, you might type data in, copy and paste, or drag and drop. You might think that shouldn't make a difference, but it does. Sometimes input validation rules are triggered by one input method but not another. So try the heuristic Violate Data Format Rules with different input methods.

Similarly, the way you navigate a GUI can make a big difference. You might use shortcut keys or the mouse. For example, you might exit a window by clicking a button or by using a shortcut key. Just as with the input method you use, the style of navigation you use can make a big difference in how the software behaves.

4.4 Variables! They're Everywhere!

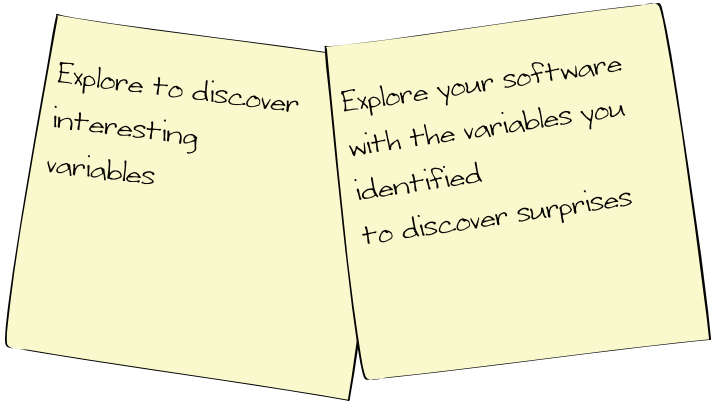
When you start looking for variables, you inevitably notice that they're everywhere. If you already felt as though you had too much to explore and too little time to explore it in, this may feel a bit overwhelming.

One programmer who saw me explaining the concepts in this chapter later told me that he found the talk thoroughly depressing. He said he was overcome by a sense of futility. He realized that no matter how much he explored, there would always be more he could explore.

If you've reached this point in the chapter and are feeling as though exploring is an impossible, never-ending task, take heart. Recognizing that there is always more to test is good news, not bad. It means that you're avoiding the pitfall of complacency. You are becoming a more effective explorer, finding new paths to cover. (Also, you might want to peek ahead to Chapter 13, *Integrate Exploration Throughout*, on page 133, to reassure yourself that there are actually good ways to limit your explorations.)

4.5 Practice Sessions

The best way to become adept at recognizing and manipulating subtle variables is to practice doing it. So that's what your practice charters are about.


Explore to discover interesting variables
Explore your software with the variables you identified to discover surprises

Remember when you're executing your charters to look for the following:

- Things you can count and apply the Zero, One, Many heuristic to
- Things you can select and apply the Some, None, All heuristic to
- Things where position matters and apply the Beginning, Middle, End heuristic to
- Things you can move to a different location
- Things with different formats that you can try the Violate Data Format Rules heuristic on
- Things that live in a hierarchy or structure where you can vary the level and nest them deeply
- Hidden settings you can manipulate
- Opportunities to change timing by increasing frequency or extending wait times

CHAPTER 5

Evaluate Results

So you start exploring, and almost immediately you run up against a conundrum: How do you know if the behavior you're seeing is correct?

Sometimes the expected results are obvious given the requirements. More often, assessing correctness involves a subjective judgment call or requires specialized knowledge. For example, consider these scenarios:

- You are exploring a web-based application with a rich, complex user interaction model. You attempt to perform basic browser actions. Is it OK that you cannot bookmark that particular web page?
- You are working on a charter to explore the system's error handling. You interrupt the process, disconnect the network, and do other generally evil things. The software handles some cases but not others. How do you distinguish between expected badness and real problems, particularly when exploring extreme negative scenarios?
- You are exploring software that performs complex calculations based on formulas devised by a team of PhD-level research specialists. You don't happen to be one of those specialists. You enter a set of values; you see a computed result. Is it right? Is it even reasonable? How can you know?

Situations like these are one of the central challenges of exploratory testing. By its very nature, exploring involves asking questions of the software that no one has thought about before. Yet you need to find a way to determine correctness or at least find a way to sift through all your observations to highlight the most important things requiring additional investigation.

In this chapter you'll discover a set of techniques for assessing the behavior of the system even when you don't know exactly what to expect. You'll discover how to identify the intransigent rules that the software must obey in all situations: the things it should always or never do, regardless of the configuration

or the users' actions. You'll learn how to identify alternative resources for expectations to compare your observations against. And you'll learn how to approximate correctness in cases where determining an exact result would be impractical.

5.1 Never and Always

Systems have rules. If you are working with accounting software, the accounts must always balance. If you are working on an editor, you might have a rule that users should always be able to undo the last action. If you are working on centralized server software, you might have a rule that there should be nothing one user could do that would cause the system to become unavailable to other users. Whatever the rules are, every system has them. It's your job to find out what the Never or Always rules are for the software you're exploring.

If at any time your explorations cause the system to violate a Never or an Always rule, it's a serious problem. So while you may not know exactly what the expected results are, at least you know what they are not. There should be nothing you could do, no combination of actions, configurations, or data that would result in one of these inviolate rules being broken.

So how do you know what the inviolate rules of your system are? One of the best places to get a sense of the kinds of things your system should never or always do is in requirements discussions. In particular, think about these factors:

- Any core capabilities of the system that must always function
- Other crosscutting "-ilities" of the system besides capability that must never be compromised: quality criteria such as reliability, scalability, usability, or accuracy
- Other risks the stakeholders are concerned about

This section helps you determine what the Never and Always conditions are for your system.

Core Capabilities

Systems exist to perform their core capabilities. Software might have bells and whistles, nice-to-haves, and filigrees that make it better, but for any given system there is a set of things that have to work even if everything else fails. These core capabilities will be unique to your system.

You may find that not everyone on a project has the same understanding of what the core capabilities of your system are. You might think that you know,

but your view might be different from the business analyst or product manager who is specifying what the system is supposed to do. To discover what your stakeholders consider to be the core capabilities of your software, try asking questions like these:

- Who will use this software and for what purpose?
- What alternatives are there, and why would someone choose this software over those alternatives?
- Can you give me the elevator pitch (a very short sales pitch) for this software?
- If nothing else were working in this system, what absolutely positively has to function?

As you listen to the answers, note any capabilities of the system that the stakeholders mention. Understanding which capabilities must always work is an excellent start to cataloging the Nevers and Alwayses of your software.

Quality Factors

The quality factors for your software, sometimes called "-ilities" or "nonfunctional requirements," give you another perspective on Never and Always. Here are some common quality criteria and example translations into Never and Always statements:

- *Accuracy*: never provide an inaccurate result; always generate error messages on invalid inputs that cannot be used in a calculation
- *Reliability*: always recover to a usable state from any user input or action, no matter how ill-timed or incorrect that input or action was
- *Availability*: always respond to user requests (within a specified time period)
- *Usability*: always give users feedback about the effects of their actions; always provide clear indicators of available actions
- *Accessibility*: always provide keyboard shortcuts; always provide text alternatives for images
- *Security*: never expose secure data to an unauthorized recipient; never execute user input as code

These are just examples; even if the key quality factors for your system are the same as those above, your Never and Always statements may very well be different from these.

Risks

The risks that your stakeholders are most concerned about are often Never and Always statements in disguise. In a safety-critical system like a medical treatment machine, key risks would involve anything that could cause harm. So the system should never put people at risk. In a financial management system, losing money is a risk. So accounts must always balance and the software must never create or destroy money. In payment systems, risks involve not getting paid or overcharging customers. That means the system should always charge for purchases but never charge twice for the same thing.

Even something as seemingly nonrisky as a game can involve risks. If the game is boring or buggy, it can alienate users. Or if users pay money for points or in-game currency, even if the sums involved are comparatively small, it's important that users not be able to cheat the system. Conversely, the system (presumably accidentally) should never cheat the users.

To discover what kinds of risks are most scary to your stakeholders, you can play the Nightmare Headline game (see Section 2.5, *The Nightmare Headline Game*, on page 21).

5.2 Alternative Resources

In the absence of detailed requirements or a specification, it might seem impossible to figure out what the expected behavior of the system should be. However, you have one of the best resources available to determine expectations: the software itself. You also have access to other resources. Depending on the context, you may be able to gather information about other, similar software packages. Or your software may be governed by publicly available standards.

Internal Consistency

Users of your software have every reason to believe that it will be self-consistent, so the software itself can help you evaluate the correctness of the behavior you observe.

Consider:

- While exploring an application, you find that you can perform an action from one screen but you cannot from another related, almost identical screen.
- While exploring an API, you find that two different calls take similar parameters but in a different order.

- While looking at a log file, you find that two different but similar logged events have an entirely different format in the log.

In each case, a user of the system would reasonably expect consistency, and you should as well.

Standards

If you are working on software related to healthcare, financial services, defense, or other regulated industries, you are probably already intimately (perhaps even painfully) familiar with the standards that your software must follow. Healthcare solutions in the United States must comply with the Health Insurance Portability and Accountability Act of 1996 (HIPAA). Similarly, if you are working on financial software for the US market, you may be governed by Sarbanes-Oxley.

Among other things, both HIPAA and Sarbanes-Oxley include requirements for encrypting personal or sensitive data. Thus, even if you do not have a requirement related to encryption, you might still explore to discover ways in which sensitive information is stored or sent in cleartext. In fact, this could become another Never for your system: sensitive information must never be stored or transferred in cleartext.

Even if you are not in a regulated industry, you may find standards that help you develop a checklist of things to watch for while exploring.

- If you are working on applications with a graphical user interface, operating system publishers like Apple and Microsoft publish user interface guidelines for their platforms.
- If you are working on software that implements communication protocols, the IEEE publishes standards for things like IPv4, IPv6, DHCP, and so on.
- If you are working on a web-based application, the W3C publishes standards for the web: HTML and CSS syntax, HTTP return codes, and so on.[1] It also offers an online syntax checker for HTML and CSS. In addition, the W3C's guidelines for accessibility for web-based applications is a handy guide to the affordances web applications need in order to work for people with disabilities.[2]

1. http://www.w3.org/
2. http://www.w3.org/standards/webdesign/accessibility

- If you are working on software where security is a concern, the *Guide to Building Secure Web Applications and Web Services* (aka "Development Guide") from the Open Web Application Security Project (OWASP) is an excellent resource for assessing the security of an application.[3]

Of course, whether or not your software is intended to comply with a given standard is an important question to consider during requirements discussions. It does not make sense to spend time and effort investigating whether the software conforms to a given standard if it was never within the scope of the project to achieve that compliance. So before you use a standard as a reference for expected behavior, make sure your stakeholders agree that the standard is applicable.

Comparables

If your company produces multiple software packages, you can use those packages as a basis for determining what the software should or should not do in a variety of situations. How should the system respond to requests to perform a secured action when the user is not authenticated? What should happen if the file submitted to the batch process is empty? How should the API respond when called incorrectly? Your company's other software may have answers that would apply to your software.

In addition, if you are working on software for consumers, you may have access to similar or related software that can serve as a reference point. Even if no ratified standards exist, standard approaches are common across a wide range of applications. Such comparables are particularly relevant if users are likely to have experience with one or more comparable systems.

Retrieving a lost password on a website is a good example. Although not a formal standard, the common practice is to include a link in an email sent to the email address on record for the user. The emailed link contains an encrypted token that serves to authenticate the user. Other (nonstandard) approaches (particularly the approach of sending the user's password in cleartext in an email) tend to garner public derision.

So as you begin exploring, see what other software is comparable to what you're working on, and let that be a guide for you.

5.3 Approximations

Some software defies any attempt to predict the exact behavior it should exhibit. Scientific and financial software often perform immensely complex

3. https://www.owasp.org/

calculations with numerous input variables. Computing the expected result is not feasible because the domain is so complex. Even subject matter experts would be hard-pressed to determine if a given result is correct.

Simulations and modeling software can be even more difficult to evaluate. They are often intentionally unpredictable, using nondeterministic algorithms to generate probabilistic results. Some games fall into this category as well: they randomly generate a board or a map, or they randomly spawn nonplayer characters.

The difficulty in determining expected results in cases like these is often called the "oracle problem" (in reference to an oracle as a source of truth rather than to the technology company). Yet there are techniques you can apply to determine whether a given result is approximately correct. You can evaluate the characteristics of the result rather than the exact values. You can feed the result into another algorithm. You can design experiments specifically to make it easier to determine if results are right or wrong. Let's look at each of these strategies in greater detail.

Evaluate Against a Range

For a moment, consider an absurd example. You are working on software that computes the airspeed of a bird—a swallow to be precise. The algorithm takes into account the bird's size, age, wingspan, and subspecies (African or European), as well as its altitude.

You try a set of values chosen specifically to explore the error handling: 0 grams, 0 days, 0 cm, European, 0 meters. The computed result is -5 meters/second. This is obviously wrong.

You plug in a more reasonable set of values: 21 grams, 4 months, 33 cm, European, 8 meters. The computed result is 12.8 meters/second.

This result isn't obviously wrong. So how will you determine if it is correct? Unless you are an expert in estimating the airspeed velocity of swallows, you probably can't. However, you can determine whether or not the answer is reasonable from readily available sources.

If you look up the maximum speed of a swallow on Wikipedia, you'll find that it says swallows have "a speed estimated at about 11 m/s, up to 20 m/s."[4] The number 12.8 meters/second passes the reasonable test. So although you cannot know if 12.8 meters/second is not right, it is not demonstrably wrong. Yet you know that providing absurd numbers produces an obviously wrong

4. http://en.wikipedia.org/wiki/Barn_Swallow

result. So using the speeds cited on Wikipedia, you explore to find combinations of reasonable values that produce a number below 0 or above 20 meters/second.

Evaluate Characteristics

One of my favorite interview questions to ask testers is this: "You are testing an API that has a random function that produces an integer between 0 and 9. How would you test it?" Anything that produces randomized results is a great example of a domain where you cannot know the expected result in advance. If you can predict the number that the random number generator will produce, it's insufficiently random.

The only way to evaluate the correctness of the random number generator is to assess the characteristics of the results after many, many uses. You can look at the distribution of and patterns in the results. You can tell there is a problem if you run the program a thousand times and 35 percent of the results are 9 while none of them are 0. Similarly, if you run the program twenty times and see the numbers 0–9 in sequence twice in a row, there is a problem. Both events are sufficiently improbable that they point to a random number generator that is not dependably random.

Invert the Result

Sometimes the easiest way to tell if a result is correct is to take the result and perform the reverse operation on it. My friend Doug Hoffman offers the example of testing a square root function. Instead of calculating the exact expected result of the function, he squared the result. Similarly, if you are exploring a translation program, translate the translation.

Another friend, Harry Robinson, gives the example of a routing algorithm. If you map the route from destination A to destination B and then from destination B to destination A, the distance for each route should be close. If the result is more than a few miles off, there's probably a serious issue.

Selecting Conditions

I was once testing a simulator that enabled users to explore what-if scenarios for a workflow. It allowed the user to define input variables to set things like the number of queues and the lead times for each queue. When the user pressed the Go button, the simulator showed the flow. As the simulation progressed, you could watch items navigate through the queues. The simulation's progression was determined entirely by the initial starting values using some really complex math.

Not being an expert in queuing theory, I could not possibly determine if the simulation correctly modeled reality. However, I had complete control over the starting state of the simulation. After tinkering with the simulation enough to understand at least a little about the effect each of the parameters had, I turned all the dials to the maximum: maximum number of queues, lead time, and so on. I watched the simulation. Then I ran it again with all the values set to their minimums. Then I set some to their maximum and others to their minimum. In choosing values at the upper end of the allowed ranges, I maximized the effects to make it easier to see the effects of the parameters. As a result I was able to ask much better questions and identify potential problems much more effectively.

5.4 Practice Sessions

Evaluating whether or not the software is behaving as intended is an exercise in judgment. You can hone your judgment by using it. The best way to do that is to practice applying the ideas in this chapter to understand what to expect from your software.

- *Identify the Never and Always conditions*: Work with your stakeholders to make a list of things that should never happen and should always happen in your system. In particular, focus your discussion around the risky conditions that would devalue the system in some way.
- *Identify external resources*: Seek out external sources of information that are relevant for your context. If you are working on low-level infrastructure software, you might seek out the related industry standards. If you are working on software that is subject to legislation, you might seek out sources that explain the legislation and its implications. Even if you are working on consumer software, you can still draw on external resources:

seek out public reviews of comparable software so you know what users do and do not like.

- *Identify useful approximations*: Look for parts of your software where you are unsure how you might calculate whether or not the software is giving you the right answer, and try applying the approximation strategies as a way of gauging correctness.

Part II

Adding Dimensions

CHAPTER 6

Vary Sequences and Interactions

Real-world users are not, on the whole, an orderly lot. They do things willy-nilly, not respecting the flow of actions that the designers intended them to follow in any given software application.

When using a Web application, people have a distressing tendency to use the Back and Forward buttons in the browser without regard for whether or not a web application supports navigating through the browser history. They don't always bother to log in before clicking on a bookmark for a page that requires authentication. Given an Undo and Redo capability, users undo then redo a series of edits any number of times. They copy and paste contents from one program into another without regard for whether or not the programs support compatible formats. They try to work around constraints that get in the way, like entering a space in required fields if they don't want to enter real data.

Worse, users sometimes allow their children or pets access to the computer, resulting in even less-logical actions that involve mashing keys, deleting stuff randomly, and turning off computers at just the wrong moment.

To discover potentially serious problems involving the use and abuse of your software, you need to vary the way you interact with it. Instead of following a sensible sequence of actions, take steps out of order. Instead of navigating using the same mechanisms every time, vary the paths you take through your system. Instead of using reasonable data, use unreasonable data.

In this chapter you will discover some ideas for shaking up sequences and interactions.

6.1 Nouns and Verbs

It's likely that if you are accustomed to the software you're exploring, you have fallen into some habits with respect to how you interact with it. Perhaps you generally use the same data, like "123 Main Street" for an address. Perhaps you have habits around navigating GUIs, like using hotkeys instead of the menu. Or perhaps you tend to run through the same sets of sequences without varying them, like logging in, setting up the data, and then searching on the data you just set up.

Habits are hard to break for two reasons: you might not even be aware that you are acting out of habit, and even if you are aware of your habits, they're comfortable. However, to explore effectively you have to go where you haven't gone before. One way to break out of your habitual rut is to allow randomness to guide you. The "Nouns and Verbs" technique provides a way to introduce randomness into how you interact with your software.

The first step in the technique is to identify the nouns and verbs of the system. Imagine you're working on an email client. The nouns, or things, in your system might include *emails*, *attachments*, *contacts*, *accounts*, and *folders*. The corresponding verbs, or actions, might include *create*, *send*, *edit*, *forward*, *copy*, *delete*, and *move*.

Although email clients are user-facing applications with a graphical interface, this technique applies to absolutely any kind of software. If you were working on embedded robotics software, the nouns of the system might involve *sensors*, *motors*, *parts*, and *targets*, while the verbs might involve *sensing*, *moving*, *turning*, and *manipulating*. All systems have nouns and verbs. Email is a convenient example because just about everyone is at least a little familiar with the domain, but this technique is good for far more than user-facing GUIs.

To vary the way you interact with a system, you can generate random scenarios by choosing nouns and verbs at random and then using that random selection to guide your actions as you use the system. You start by making a list of nouns and a list of verbs from which to pull. For the email example, your lists might be like those in Table 1, *Example nouns and verbs for an email client*, on page 61.

These two lists represent just a small subset of the possible nouns and verbs you might identify. If you were to spend time brainstorming, you could certainly generate much longer lists. However, even these short lists offer enough fodder to prompt a vast number of possible scenarios.

Nouns	Verbs
Message	Send
Header	Receive
Draft	Edit
Attachment	Save
Signature	Export
Folder	Move
Recipient	Archive
Appointment	Reply
Note	Forward
Contact	Delete

Table 1—Example nouns and verbs for an email client

If you randomly selected an item from each of these two lists and put them together, you would have a hundred possible unique combinations of actions and targets for those actions. (You could use something as simple as dice, cards, or a spinner to do the random selection.) If you then randomly chose ten noun/verb combinations, you would have a googol (10^{100}) possible sequences. With so many possibilities, your generated list of steps is almost guaranteed to ensure that you will take actions in a very different order than you otherwise would.

Some of the noun and verb combinations would be completely sensible, like "Delete Contact." Other combinations might be nonsense, like "Archive Header." Such nonsensical combinations might seem unusable, but they're even better for inspiring variations in usage than the combinations that make sense. When you come up with a combination that does not make sense, you have to invoke your creativity. What could archiving a header mean? Perhaps it means archiving an empty email that is nothing but headers. Perhaps it means copying the contents of an email header into some other document to save it. Perhaps there is some action, like exporting, that the email client allows you to take on a header. Searching around for an interpretation for the nonsensical combinations can put you on a path to unexplored territory in your software.

Just like nonsensical noun/verb pairings, illogical sequences push you into creative interpretations. If you randomly drew "Send Message" immediately followed by "Add Recipient," you might wonder how to add a recipient given that the message has already been sent. Perhaps you would send a message

and then fish the message out of your Sent folder and forward it to an additional recipient.

By randomly selecting noun/verb combinations in sequences, you'll find yourself tromping through new territory in your software in no time. As you work your way through your randomly generated scenario, remember to watch carefully for hints of unexpected behavior, and be particularly alert for any violation of the Alwayses and Nevers for your system.

6.2 Random Navigation

Imagine you are exploring a GUI-based application. Do you intentionally and consciously mix up how you drive your way though the application? Or do you drive by instinct, reaching for the mouse or keyboard out of habit?

Changing up the way you navigate through a GUI is important. If you never use a mouse, you won't notice any surprises related to clicking or hovering over controls. If you never navigate a web application using the browser history, you won't notice surprises related to clicking the browser's Back button. If you never copy and paste instead of typing, you won't notice if copying and pasting values fails to trigger the validation rules in a field.

As you navigate a GUI, take note of all the ways you can accomplish the same action. Notice the GUI elements you can manipulate, including buttons, links, menus, and window controls. Assemble a list of keyboard shortcuts, including both application-specific shortcuts and common shortcuts for your operating system. Experiment to find all the ways you can enter data. Try typing, autocompleting, copying and pasting, and dragging and dropping. Notice where GUI mechanisms for entering or selecting values occur, such as spinner controls on numerical fields or elements you can double-click or tap to select. Make note of the parts of your software that allow you to undo or redo changes.

With your list of navigation options in hand, explore your software while considering these questions:

- What are all the ways you can close the current window?
- What are all the ways you can go back from your current location to the previous window or view?
- What are all the ways you can input data into a given field?
- What are all the ways you can submit or save changes?
- What are all the ways you can cancel or undo an action?

As you explore, make a point of choosing different mechanisms for each of these kinds of actions.

6.3 Personas

Alan Cooper first captured the idea of personas in his book *The Inmates Are Running the Asylum [Coo99]*. As he describes it, a persona is an archetype for a class of users our system needs to serve. Alan gives his personas names and personality types. It's not enough to say that a user is an administrator. The user is Pradeep, a 34-year-old Microsoft Certified Professional.

Just as personas are useful for designing systems, they're also useful for exploring systems. Adopting the mantle of a persona prompts you to interact with the software with a self-consistent set of assumptions, expectations, desires, and quirks.

Imagine you're working on software for a business. How would these personas interact with the software differently?

> Charles is a 40-something supervisor and has worked at the company for twenty years. He hopes to work there the rest of his life. However, Charles has not kept up with changes in technology, and he knows it. He's concerned about what his lack of technical acumen might do to his career, so he tries to hide it. That was easier some years ago, but these days software has invaded every aspect of the business. Charles has succeeded in his job because he's driven and brings the best out in his staff. His staff loves him because he's a good leader, but he's not a patient man.
>
> Jaina is a 20-year-old in her first position. She's been using computers since before she can remember. Although Jaina is quite competent technically, she lacks work experience and is easily confused by company procedures and jargon. However, she is eager to learn and asks a lot of questions, sometimes interrupting her work to seek out answers. She is also keen to show off what she can do.
>
> Boris is 50-something and loves technology. He's always at the bleeding edge. When tablet computers appeared, he was the first to bring one into the office. He insists on upgrading his software and applications the minute new versions are released. Boris also understands the business inside and out. He's worked in the field for thirty years, eight of them at this company and the previous twenty-two years for competitors.

As you can imagine, Charles, Jaina, and Boris will all interact with any given application in completely different ways. Each of them will have a different comfort level with technology in general and the company's custom software in particular. Each of them will use it for different purposes. And each of them will exercise it in different ways. Because Charles is both impatient and more than a little irritated by technology he doesn't understand, he's likely to be a keyboard banger: if the computer is slow to respond Charles is likely to press keys again and again until the computer responds. By contrast, Jaina is likely to use her computer skills to speed up her work: if there is a back door

or a way to script the software to automate basic processes, Jaina will find it. Finally, Boris is likely to stumble over issues related to newer technologies.

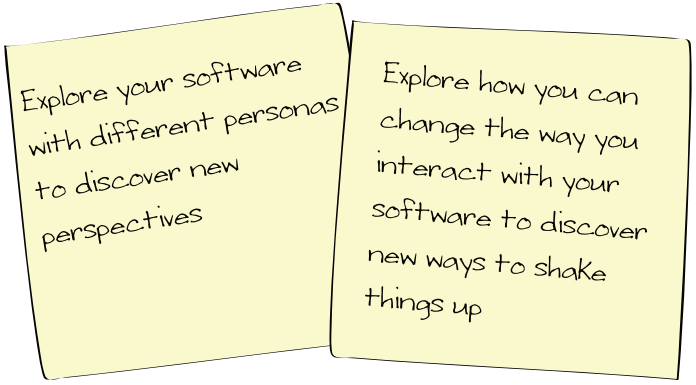
Put yourself in the shoes of your persona. Adopt those personality traits, intentions, and concerns as you explore the software. Try to accomplish the same tasks your persona would need to accomplish.

By defining archetypical personas to take on as characters while exploring your software, you're more likely to discover issues related to variations in how different users interact with your software.

You can take this idea one step further and adopt extreme personas—caricatures of users. Take Charles, for example. He's not technical. Exacerbate that characteristic: create a persona that is every tech support engineer's worst nightmare. Imagine someone who is so un-technical that he thinks the mouse is a foot pedal or doesn't understand why he can't send email if his network connection is down. Now be that person. Notice the kinds of things you try to do, how the software responds, and how you react. This may require a great deal of imagination on your part. Think of it as a form of improv. You're playing an absurd character; run with it and have fun.

6.4 Practice Sessions

Use the ideas in this chapter to shake up the way you explore your software. While in a session, notice your habits. Do you tend to navigate with a keyboard, or do you tend to use a mouse? Do you tend to use the same data over and over? Do you tend to do things in the same order?


Explore your software with different personas to discover new perspectives
Explore how you can change the way you interact with your software to discover new ways to shake things up

Whatever habits you notice, make a conscious effort to avoid doing things the way you normally do. If you always use keyboard shortcuts, slow down and use the mouse. Change up the data you use: make it more realistic or

more likely to expose problems (for example, the last name *O'Malley* is often a good choice).

Finally, choose a persona to adopt while doing all this. Act in a way that is consistent with how you imagine that persona would act. You might choose an extreme persona, perhaps even a cartoon character. How would Fred Flintstone use your software? What about Homer or Bart Simpson? Or Sponge Bob Square Pants?

CHAPTER 7

Explore Entities and Their Relationships

Things. Your software uses things, depends on things, and manages things. It lets users create, update, and delete things. Things are the reason your software exists.

What are these things? Anything. They could be things common to just about any system, such as user accounts or reports or messages. They could be things specific to the domain in which you are working, such as invoices for accounting systems, items in inventory systems, nodes in network management systems, or friends in social networks.

These things are nouns, as discussed in Section 6.1, *Nouns and Verbs*, on page 60. They're also known as *entities*, a term that has deep roots in structured analysis and design.

In this chapter you will learn how to identify the entities in your system and map their relationships. You'll learn how to explore using the CRUD (Create, Read, Update, Delete) heuristic. You'll also learn how to combine the heuristics for exploring with entities with the heuristics from previous chapters, such as 0, 1, Many.

7.1 Recognizing Entities, Attributes, and Dependencies

As soon as you start looking for entities within your system, you'll find them.

You can most readily identify entities by looking for things you can create. For example, if you are working with a GUI application, look for any place where you can create new objects by choosing New from a menu or by clicking on a plus icon. If you are working with an API, look for instances where you call a constructor.

Entities have *attributes*, data elements that describe some aspect of the entity. An email message has a sender, a subject, and a body. An inventory item has

a name, a description, and a price. As you identify entities, list their attributes. Varying attributes (see Chapter 4, *Find Interesting Variations*, on page 33) as you explore with the heuristics in this chapter is a key to deepening your investigation.

Finding Nonobvious Entities

Although some entities will be obvious even to a casual observer, your system most likely has other kinds of entities that are less obvious at first glance. As with the other techniques in this book, the approaches in this chapter work best if you look past the superficial to find the more subtle opportunities for exploration. To that end, here are some approaches for identifying less conspicuous entities:

- Be alert for entities that are more visible to the system than to users. Logging into a system typically spawns a session. That session is an entity in its own right. It is created when you log in and destroyed when you log out. While it exists, it is likely to have attributes such as duration and connecting IP address.
- Look for GUI forms, even if they are not explicitly associated with creating a new record. Any time you have to fill in data in fields on a form and then submit it, you are probably creating or updating an entity of some kind.
- Back your way into identifying entities by identifying attributes. Any data that your system manages is connected in some way to some kind of entity. Choose some small piece of data, then look for some larger thing that piece of data might be associated with. Be particularly alert for attributes represented graphically rather than through text. Formatting options such as position, size, and color can be good cues that indicate you've found another kind of entity.

As you expand your list of different kinds of entities in your system, you may start to notice that they're all related. Indeed, something that seems initially like a simple attribute of one entity may itself be an entity. An email address is addressed to someone. That address could be a simple attribute, such as a value in the To field. Or it could be a contact with a name, an email address, a street address, and so on.

Once you've identified a preliminary set of entities, the next step is to map out how the different kinds of entities are related to one another.

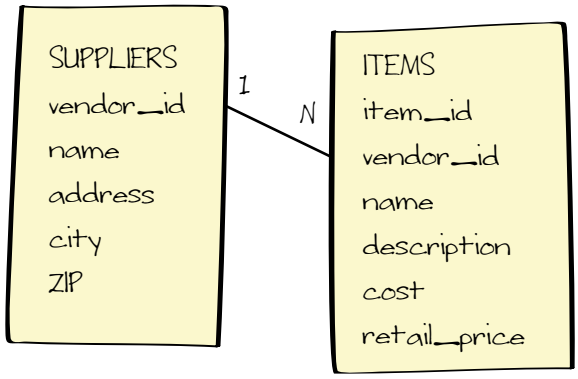
Mapping Relationships

Entities almost never exist in isolation. Things have a relationship to one another. A thing might depend on another thing. A thing might belong to another thing or contain many other subthings. Everything is connected. In an email client, messages are associated with other messages in conversation threads. They might also be connected with contacts in your address book.

You can recognize dependencies by paying attention to how you talk about the things in your system. Any time you would say that an entity "has a...," you have probably identified a relationship between two entities. An account has a primary contact. A statement has transactions. A document has revisions.

You can use an *entity relationship diagram* (ERD) to describe the nature of these connections. Such diagrams have been around for decades. It may be more common to see an ERD as a system design artifact created by a programmer, database administrator, or architect while drafting a technical specification. However, they're just as valuable for analyzing systems for testing.

Let's take an example. Imagine you are working on an inventory management system. Each item in the system comes from a supplier, and each supplier has multiple items. You could express the relationship between these entities with a picture like the one here:



The two boxes in the picture represent the entities—suppliers and items—and the connection between them. The lists inside the boxes include the different attributes associated with each entity. The small "1" and "N" on the line connecting the two boxes shows that for each supplier in the system there could be many items.

As you sketch the relationships between entities in your system, you may find that they are more complicated than this simple one-to-many relationship. For example, what if a given item is available from multiple suppliers? Each supplier would set its own price for the item, but your company might still use the same stocking number as an identifier, no matter which supplier provided the part.

If you were designing a database schema to represent these entities, you would need to add another table between the two in order to provide a way to make this many-to-many connection. The new table would make the connection between suppliers and items, providing a place for each unique relationship between a supplier and an item.

Such a relationship might look like the entity relationship diagram in Figure 1, *A many-to-many relationship: many suppliers and many items*:

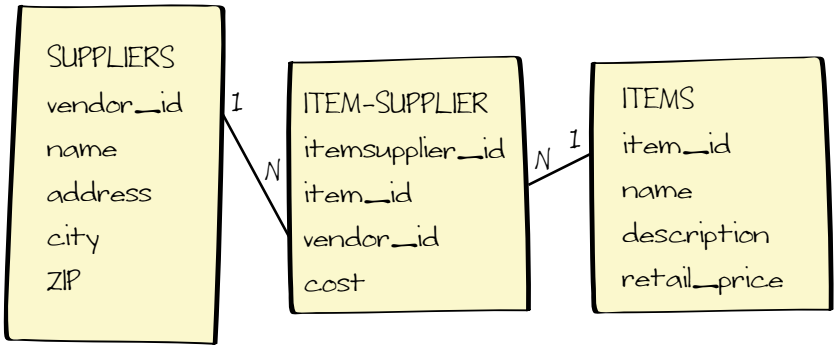


Figure 1— A many-to-many relationship: many suppliers and many items

When you are sketching your entity-relationship diagrams, you may not have insight into exactly how the data is stored, but you can make an educated guess. Then having sketched the connection between the entities, you can use the model to suggest interesting paths to explore. The remainder of this chapter introduces heuristics for exploring with entities.

7.2 CRUD: Create, Read, Update, Delete

The CRUD heuristic stands for Create, Read, Update, Delete. It comes from the world of relational databases, where those four actions represent the core operations you can perform on data in a table. When a software application

exists primarily as a mechanism to get data into and out of a database, sometimes people will describe it as a "basic CRUD application."

Although the CRUD heuristic has its roots in relational databases, it applies to any kind of system regardless of the kind of storage the system uses. Broaden your exploration by trying variations while creating, viewing, updating, and deleting each of the different kinds of entities that your software manages.

In some systems each of the four basic actions involves an explicit and straightforward action. For example, you create a document by selecting New from the Document menu. Sometimes, however, it is not obvious how to do all four CRUD operations. Sometimes the way to create, update, or delete an entity involves an indirect action. If you're working on an email system, email messages are entities. If you send an email, you've created a message. What may be less obvious is that from the point of view of the email client software, receiving an email is also a way of creating a message.

If that's all there were to using CRUD, it would be just another aspect of Chapter 6, *Vary Sequences and Interactions*, on page 59. The next step, however, is to explore deeper by considering how entities relate to each other and then exploring variations in those relationships in combination with both CRUD and other heuristics.

CRUD with Data Variations

As discussed in Chapter 4, *Find Interesting Variations*, on page 33, a good starting point for exploring is to vary values. When you begin exploring entities with CRUD, try creating, viewing, updating, and deleting entities while varying values in their attributes.

Sometimes even simple variations yield surprises. In one software application I tested, a user account had a URL associated with it. If the user omitted the "http://" when specifying the web address, the software appeared to save the data but didn't. This is a trivial variation in the format of a web address, yet it revealed a nasty bug.

A common pattern of misbehavior in software involves attempting to clear optional values. I see an example of this regularly in the mail that the post office delivers to me. I currently live in a house. In the past, I have lived in apartments. Occasionally I receive mail addressed to my house but with my last apartment number also displayed in the address. The company clearly updated my contact information, but it was somehow unable to blank out my old apartment number. I have seen similar problems with software I have

tested. So one of the experiments I always run is to create entities with values in all the optional fields and then attempt an update where I delete the values in those fields.

All the Ways to CRUD

Frequently there are multiple ways to CRUD an entity. For example, with a desktop application, you can create a new file by selecting New from the File menu or by choosing Save As on an existing document. Look for different ways to CRUD entities, such as these examples:

- By navigating through different paths, screens, or commands
- As a side effect of some other action
- Through undo/redo commands

Watch for differences in the attributes or behavior of entities created one way as opposed to another. For example, in some inventory systems, if the inventory level dips below a threshold, the system automatically generates an order. That order may go through a different workflow than an order that is explicitly created by a user. Such differences can yield rich territory for exploration.

CRUD with Zero, One, Many Dependents

Understanding how entities in the system relate to one another means that you can combine the CRUD heuristic with the Zero, One, Many heuristic. Consider these experiments:

- Can you create an entity with no dependents? For example, if you were working with an inventory management system, can you find a way to create an item that has no supplier? This is particularly interesting if you are able to make the system violate a business constraint.
- What happens if you delete something that has many dependents? Are the dependent entities also deleted or affected in any way? In the example inventory management system, what happens to items if you delete the supplier from the system?
- Can you update the entity so that it is associated with a different dependent? Again, in the hypothetical inventory management system, can you update an item so that it's associated with a different supplier?

In several systems I have tested, if you delete a parent entity, like a supplier, you can no longer view any of that entity's dependents. Any attempt to view

the dependent entities results in some kind of error—sometimes an ugly stack trace.

Wherever you have a more complicated relationship between entities, as with the relationship shown in Figure 1, *A many-to-many relationship: many suppliers and many items*, on page 70, you have even more opportunities to apply the 0, 1, Many heuristic along with the CRUD heuristic. You could try creating the condition where a supplier has zero items by deleting a supplier's last item. You can also try creating a condition where an item has zero suppliers by deleting an item's last supplier. And you can create a condition where a given supplier has one and only one item, and that item comes from one and only one supplier.

As you try cases like these, remember the advice in Chapter 3, *Observe the Details*, on page 25: watch consoles and logs to detect hidden problems. Also watch other places in the system. You never know when creating conditions like these will trigger problems elsewhere in the system, such as in reports or with searches. This leads us to the next heuristic, Follow the Data.

7.3 Following the Data

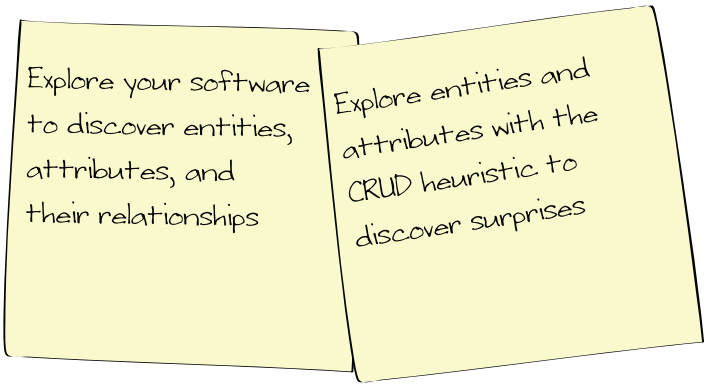
Entities have a life cycle. They are created and eventually deleted. In between their births and deaths, they might show up in a number of places in your system. You can see lists of entities in searches and reports. Depending on the nature of the software you are testing, there might be other kinds of views that show entities, such as a queue in a workflow or a user's activity log.

As you explore different ways to CRUD entities with a wide range of attribute values and associated dependents, seek out these alternate places where you can see entities. Perform actions on the entities through these views, if you can. Watch for any indication that a change made in one part of the system is not handled well in another part of the system.

Also watch for side effects that manifest in one area of the software as a result of actions taken elsewhere. For example, if you work with a desktop publishing application, you've probably noticed that you can open a document from the Most Recent Documents list. If you delete the document, is it still on the Most Recent list? What happens if you try to open it?

7.4 Practice Sessions

Of course, the best way to practice the ideas in this chapter is to use them.


Explore your software to discover entities, attributes, and their relationships
Explore entities and attributes with the CRUD heuristic to discover surprises

So explore your software to discover entities, attributes, and their relationships. Sketch a model of the entities in the form of an entity relationship diagram. Use the diagram to help you explore using the CRUD heuristic. Combine the CRUD heuristic with heuristics for varying the values of attributes. Follow the data through the system to discover if entities look or behave differently in different views.

As you do so, pay attention to how the concepts in Chapter 4, *Find Interesting Variations*, on page 33, apply to entities. What variables do you notice? Attribute values can vary, so they're variables. There are other variables as well, such as counts of entities and counts of children.

Notice also that the ideas in Chapter 6, *Vary Sequences and Interactions*, on page 59, apply here as well. Entities are nouns: creating, reading, updating, and deleting are all verbs.

You can deepen your exploration further by combining the ideas in all three chapters: vary sequences of CRUD actions with entities having various attribute values or other properties.

CHAPTER 8

Discover States and Transitions

Have you ever encountered a failure that was extremely difficult to reproduce? Perhaps you've seen a catastrophic error that happens only sporadically, or maybe you stumbled on corrupted data and could not trace the root cause.

Such defects are often triggered when something happens during a brief window of vulnerability: a moment in time when all the conditions line up just right so something can go very wrong. A file happens to be locked just when the software attempts to write to it. Your session times out just at the moment you try to access secure content. A race condition exists where on rare occasions one part of the system attempts to update a record before another part finishes creating it.

These conditions are usually short-lived and hard to spot. You might not know how to cause them intentionally or even how to tell when they're open so you can take advantage of them. If you can't see the window of vulnerability, discovering or reproducing bugs related to it becomes a frustrating matter of trial and error.

Fortunately, there's a systematic approach to discovering and exploiting these windows of vulnerability using state models. In this chapter, you'll learn how to map a state model and use heuristics to surface surprises related to timing.

It's important to note that you'll get more out of this chapter if you focus on observable behavior rather than on implementation details. Modeling from an external perspective enables you to see states and responses that stem from the whole technology stack and not just from the layer of that stack with which you are most familiar. So even if you happen to have knowledge of the internals of the software, you'll need to set that knowledge aside so that you can focus on what you see happening rather than on what you know is going on under the covers.

8.1 Recognizing States and Events

Noticing state transitions and the events that trigger them is easier said than done. Sometimes the states are subtle and fleeting, triggered by the passage of time or by an event you cannot control directly. Such states are especially important to identify because they are usually the most interesting to explore, yet their transitory nature makes them difficult to see. Similarly, noticing all the conditions that trigger a shift in states requires careful attention to detail. In this section you'll find a handy guide to identifying even hard-to-spot states and events.

States Are Behavioral Modes

My friend Alan A. Jorgensen uses three simple questions to detect states:

- Are there things I could do now that I could not do before?
- Are there things I cannot do now that I could do before?
- Do my actions have different results now than before?

Whenever the questions reveal differences in behavior, Alan knows he has found a new state.

You can also detect states by attending to the language you use to describe what's happening. Watch for situations where you use the word *while* in describing the software, such as in these examples:

- "While the system is importing data…"
- "While the system is running the report…"
- "While the account is suspended…"
- "While the call is on hold…"
- "While the server collects usage data…"

Any time you can use the word *while* in describing behavior, you've identified a state.

Consider an example. Imagine you are identifying states associated with logging into the system. You are currently on the login screen, as shown in Figure 2, *The login user interface*, on page 77.

The login screen represents a state: the system is waiting for login credentials. You can't do anything interesting until you log in. You might call the current state "Logged Out."

Once you provide a username and password, you might notice that there is a period of time after you enter the login information and before you see the

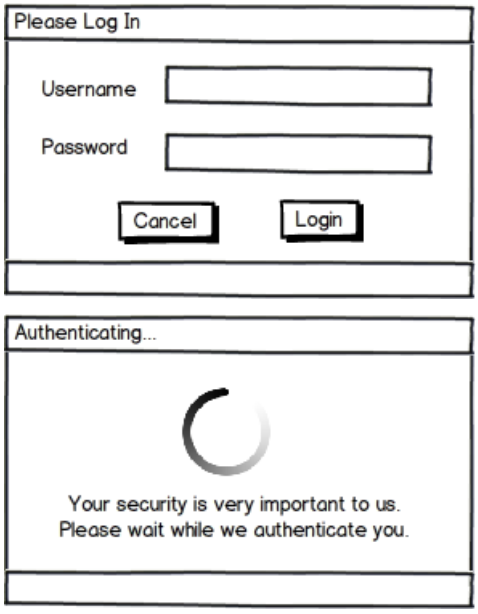


Figure 2—The login user interface

first screen of the application. There's an animated spinning wheel signaling you that something is happening.

When the spinner is on the screen, you could say "While it's authenticating…" Aha! "Authenticating" is another state. It's a transitional state that will end when the system finishes taking whatever steps are necessary to authenticate your credentials.

Events Trigger State Transitions

User actions are the most obvious kind of event: you take an action and it triggers a response in the system. In the login example above, you click the Login button and the system transitions into the authenticating state. User actions aren't confined to GUI interfaces. You might issue a command at the command line or make a server request using an API.

However, user actions are just one kind of event. Other kinds of events are harder to control and thus may be harder to spot. As you're looking for events, consider the following:

- *Externally generated events*: any event coming from outside the software. For example, if your software monitors the contents of a directory on the

file system, changing the contents of that directory is an example of an externally generated event.

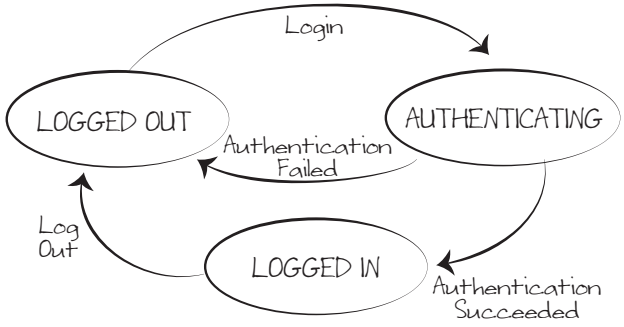
- *System-generated events*: any event triggered by the software itself. Often these events are the result of the system completing some background activity: loading or exporting data, connecting to a remote server, authenticating a user, or performing some background calculation. Any time you notice that there is a delay between your action and the system's response, it is likely that there is an interstitial state and associated system-generated event hidden behind the scenes.
- *Passage of time*: some events, such as timeouts, are simply the result of the passage of time. Look for conditions that occur after a given duration, such as alerts. Also watch for events that occur at a specific time, such as a backup scheduled for midnight every night.

Finally, remember that every state is triggered by an event. Thus, if you identify a state, back up to identify the event that triggered the state. As you are doing this analysis, you're likely to find that you switch back and forth between looking for events and looking for states since the two things are so inextricably intertwined.

8.2 Diagramming a State Model

Now that you know how to recognize states and events, you can draw a diagram of the relationship between them.

For example, a state model for the login example might look like the following. Note that states appear in circles, while arrows between the circles represent the possible transitions. The labels on the arrows are the events.



Creating a state model takes more than just noticing states and events. So many states and events exist even in a simple system that the task can quickly become overwhelming. Here are some strategies to keep you on track.

Narrow Your Focus

You can narrow the scope of your model by identifying a single target, such as a feature or workflow. Take uploading a file: the workflow involved might involve states like Selecting a File, Uploading, and Confirming Upload.

You might even choose to focus on the life cycle of an entity within the system. For example, imagine you are exploring a bug-tracking system (something that is near and dear to many testers' hearts). A given bug record is an entity. Bugs typically have an entire life cycle, with states like New, Assigned, Fixed, Deferred, Verified, and Closed. User accounts are another, different entity that might have states like Active or Suspended.

Choosing a focused target is important. Otherwise you'll find yourself suffering from analysis paralysis, identifying more and more states and transitions but not having any time left to use that knowledge to help your explorations. If you think you might not have chosen a sufficiently well-defined target, try naming it. If you cannot give it a simple name, it's probably more than one target.

Identify a Perspective

Consider a phone call. You could map states from the perspective of the caller. In that case, an event might include taking the phone off the hook to get a dial tone. Another event might include putting the phone back on the hook. If the caller pressed the speakerphone button before replacing the handset on the phone (another event), then the call will be in a different state than if the caller simply hung up the handset.

Alternatively you could map the states from the perspective of the call. The call would not come into existence until two parties are connected. Once connected, the call might be placed on hold, forwarded, or connected with a third party. The events might still involve user actions on a handset (e.g., pressing the Hold button), but the perspective dictates which states and events belong in the diagram and thus narrows the scope.

Dial Up or Down the Level of Abstraction

If you are modeling a simple small interaction, you can do so at a low level of detail. You can map out all the teeny transitory states between user-generated events. This will give you tremendous insight into those pernicious, narrow windows of vulnerability.

However, you'll need to work at a higher level of abstraction if your target is something bigger. Instead of listing out numerous separate transitory states, you'll lump them together into one big state with a less specific name.

Imagine you are modeling states associated with launching some kind of client program that connects to a server. It could be any kind of client: a point-of-sale system, an email client, a music player…anything.

If you chose a very narrow target, like the launch process, you would identify states at a lower level of abstraction. So you would look for indications of distinct transitory states between the time the user double-clicked on the icon for the client and the time the client is ready for the user to begin interacting with it. You might notice states like "Connecting to the Database" and "Caching Data."

If, however, your target is the states of the client itself, discovering all the transitory states associated with the launch would be too much detail. You'd get bogged down in analyzing fine-grained details when you really wanted to look at the big picture. So you would roll all those substates associated with the client starting up into a single state, "Launching."

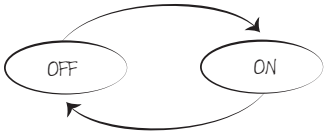
8.3 Exploring with State Models

Now that you have a state model, you can use it as a guide. You can use your map to help you identify all the ways to get from State A to State B and to identify all the events that you can trigger in an attempt to interrupt the states. For that matter, you can use it to identify opportunities to find memory leaks or other bad behavior by cycling through event loops.

All the Ways

Discovering all the ways to get from one state to another can help you identify new ways of interacting with your software.

Consider a state model that can apply to absolutely any software system anywhere:



At some point the system is Off. You boot it up, and now it is On. There is undoubtedly some mechanism to turn the system back to Off again. This model could describe anything from lights to ATM machines to web services to games.

Consider these questions:

- *What are all the ways to go from Off to On?*

You can probably launch the system manually. In addition, the system may launch automatically when the hardware boots up. There may be other mechanisms to launch the software. For example, some high reliability systems have a watchdog daemon that restarts key processes if they fail.

- *What are all the ways to go from On to Off?*

  Most systems have a mechanism for an orderly shutting down of the system, but that's not the only way to turn a system off. Killing processes, pulling the plug on the machine, closing the lid on a laptop, or timing out could be other ways to go from On to Off.

Let's look at how you might apply this technique to an audit trail feature. Imagine that the software logs an entry in the audit trail any time a user makes a change within the system. This audit trail enables users to see who changed what and when something was done.

In most systems, there are multiple ways to make updates. You might make changes one by one: opening each thing you are editing one at a time, making the change, and then saving. Some systems let you do bulk updates. For example, in an email client, you could select multiple emails and give them all the same tag. A batch import feature could enable you to do a bulk edit by putting all the updates in a file and then importing.

If you are exploring the audit trail feature by looking for all the ways you can update a record, you may discover that the audit trail isn't updated correctly for some of the update methods. In one real world case, I did exactly that and discovered that the system only logged activity in the audit trail when I edited records one at a time. When I changed the status of multiple records at once, there was no corresponding entry made in the audit trail for any of the records.

Interrupting

When you interrupt a state by triggering an unexpected event, the software may or may not respond well. For each state you identify, you might try interrupting it using one of these methods:

- Clicking user controls (Cancel buttons are particularly fun.)
- Logging out
- Letting a session time out
- Killing the process
- Closing the lid on the laptop
- Pulling the plug on the machine
- Disconnecting the machine from the network

After you interrupt the state, consider these questions:

- Did the software respond to the interruption the way you expected? Or is it now in an unexpected state?
- After you recover from the interruption, is the behavior restored to normal?
- Did the software lose or corrupt any data?

Also look for any violations of the Alwayses and Nevers for your system (see Section 5.1, *Never and Always*, on page 48).

Variables Revisited

You learned about identifying variables in Chapter 4, *Find Interesting Variations*, on page 33. Remember that a variable is anything that you can change, or cause to be changed, that affects the behavior of the system. It's worth revisiting the concept of variables in the context of a state model. For example, consider these variables:

- The previous state
- The mechanism or event used to get to the current state (such as shutting down gracefully vs. killing the process)
- The number of times the system has been in this state before (such as the number of times the system has gone through the startup sequence)
- The number of times a given event has occurred
- The number of times through a loop of connected states
- The length of time the system has been in a given state

These are just a few ideas to get you started. You could vary any number of other things. For that matter, you can combine other variables such as configuration settings, inputs, or outputs with state/event-related variables.

8.4 Changing the Representation: State Tables

If you change the representation of the state model to a table, you can find even more opportunities for unexpected transitions.

Consider my alarm clock. It looks like Figure 3, *Alarm clock interface*, and it operates like a typical alarm clock.

- You can turn the alarm on or off.

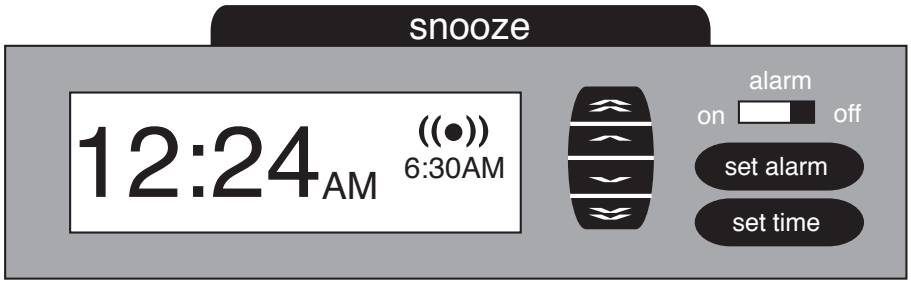
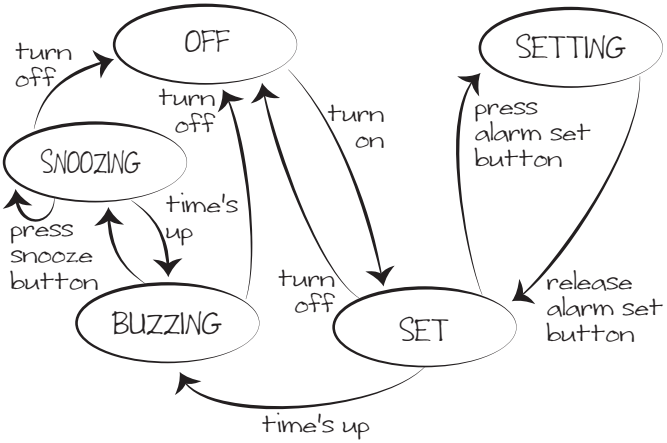


Figure 3—Alarm clock interface

- You can set the alarm time by pressing and holding the alarm set button and then pressing an arrow to indicate whether you want the alarm earlier or later.
- If the alarm is turned on, it buzzes when the clock ticks over to the indicated time.
- If you press the snooze button, the alarm quiets for nine minutes and then sounds again.

If you were to explore it, you might identify states like Off, Snooze, Setting, and so on. You could then create a state model like this:



Instead of representing the state model with circles and arrows, you can put the states and events in a table, like in Figure 4, *The alarm clock state model in table form*, on page 84. This diagram shows states down the side and events across the top. The body of the table is not filled in yet. Based on the state

model expressed with circles and arrows, we can partially fill in the values for the table. For example, if the alarm is set and we hold down the Alarm Set button, we'll be in the Setting state. So the cell at the intersection of Set and Press Set would get the value Setting.

events → / states ↓	Turn On	Press Set	Release Set	Press Snooze	Time's Up	Turn Off
Off						
Setting						
Set						
Buzzing						
Snoozing						

Figure 4—The alarm clock state model in table form

If we were to fill out the rest of the table with the information we have, it might look like this:

events → / states ↓	Turn On	Press Set	Release Set	Press Snooze	Time's Up	Turn Off
Off	Set	???	N/A	???	Off	N/A
Setting	N/A	N/A	Set	???	???	Off
Set	N/A	Setting	N/A	???	Buzzing	Off
Buzzing	N/A	???	N/A	Snoozing	N/A	Off
Snoozing	N/A	???	N/A	???	Buzzing	Off

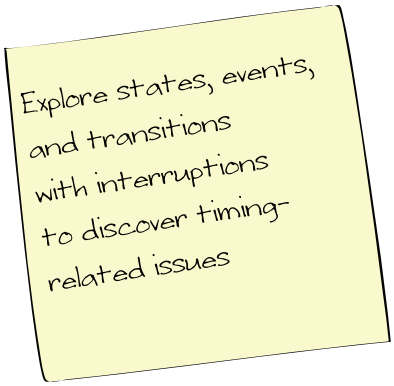
What happens if we're setting the alarm and the clock matches the current set alarm time? Does the alarm start buzzing? Should it? These are the cells where we don't know what will, or what should, happen. They contain "???" in the table. These possible but unexplored transitions are ripe for discovering surprises.

State models prompt you to consider transitions. State tables prompt you to consider what happens when you cause events to occur outside the bounds of expected transitions. As a result, state tables are particularly good for discovering problems related to interruptions or unhandled transitions.

8.5 Practice Sessions

Try modeling an area of the system you're testing. You might focus on a low-level interaction where you have to pay close attention to tiny details, as demonstrated with the login example (Figure 2, *The login user interface*, on page 77). Or you might try modeling states for something larger and at a higher level of abstraction.

Having chosen your target, explore it to discover states and events. Start with user actions: poke at buttons, change data, move things around. As you do, watch for transitory states and the conditions that trigger their transitions. Create a map of the state model.



With your model in hand, use it to suggest interesting ways to provoke the system. Apply the heuristics in this chapter:

- *All the Ways*: Take a transition between states, and find additional ways to trigger it.
- *Interrupting:* Find ways to trigger user-controllable events, such as logging out or pushing buttons, during the fleeting interstitial states or during other states that probably were not designed to handle unexpected events.
- *Variables*: Look for things you can vary. For example, the number of times an event has occurred in the past is a variable. Recall the story of the Therac-25 machine in Chapter 4, *Find Interesting Variations*, on page 33, where one of the key variables was the number of times the system had been rebooted.

- *Change the Representation*: Express your state model as a table; then consider the combinations of states and events suggested by the matrix. Are there states where you can apply events that you hadn't considered before?

Finally, the next time you encounter an intermittent failure, map the states and events in the areas around the failure. Perhaps you'll discover that the underlying cause of the failure has to do with triggering an event to occur at just the right, or wrong, time.

CHAPTER 9

Explore the Ecosystem

Software never exists in isolation. It runs on an operating system. It depends on libraries of reusable code or external services. It uses system resources like memory, the file system, databases, and network connections. It interoperates or integrates with other applications.

When you're focused on the internals of your software, it's easy to forget all these dependencies that make up the larger system. You're thinking about algorithms and business rules and user flows, not what happens if the network goes down. Yet all those external dependencies represent a wide range of places that you cannot control directly and where things can go horribly wrong. Network connectivity can be flaky. Files can be locked at the file-system level by other programs. Databases can be corrupted or unavailable. Other programs running on the same computer can hog the CPU power and memory. The cloud-based data repository can go offline without warning.

Reliable software takes such unexpected events in the ecosystem into account. It can be difficult, however, to think through all the different kinds of unexpected events in advance. In this chapter you'll learn how to explore your software within the context of its environment in order to discover conditions that trigger bad behavior. You'll learn how to play the What If? game to find situations where dependencies outside your software's control have the power to bring down your system. You'll discover how to use external interfaces to manipulate internal dependencies. You'll also revisit the Follow the Data heuristic to uncover risks at the edges of your system, where data comes in or goes out.

9.1 Diagramming the Ecosystem

The ecosystem of your system includes the environment in which your software lives, all the interfaces into your software, and all the external dependencies.

Drawing a picture of the entire ecosystem gives you a map you can use in systematically exploring around connections and dependencies to discover risks associated with the parts of your system that are outside your software's control.

The style of diagram I use to represent the ecosystem is a mashup of two traditional system analysis and design artifacts: a context diagram and a deployment diagram. A *context diagram* shows how your software connects to the world outside your system. It's all about the interfaces, users, and connections to integrated systems. By contrast, a *deployment diagram* shows where the components that make up your system, such as databases, configuration files, and executables, live in a production deployment.

Including both internal resources and external dependencies in a single diagram gives you a cohesive view of the big picture. It also makes sense to put both internal and external dependencies in the same picture because you'll use the same techniques when exploring for vulnerabilities associated with dependencies, whether those dependencies are technically inside or outside your system.

It's worth noting that although there are formalized styles for these kinds of analysis diagrams, they're all more complicated than what you need to support your exploration. For these diagrams, I will be using my own variation on the Galactic Modeling Language, GML.[1]

Step 1: Draw the Interfaces

To begin, draw a large circle in the middle of a blank sheet of paper and label it with the name of your system (Figure 5, *Starting a context diagram*, on page 89). This circle represents the boundary between internal and external dependencies. For this example, I'll diagram a web-based system I once worked on that took credit card payments.

The people who use your software are outside the system. Indicate the user interfaces with connections to people. In the case of my example, we had two kinds of users: regular end users and administrators. Each had its own user interface (Figure 6, *Adding the interfaces*, on page 89).

User interfaces are just one kind of interface. Your software may have public interfaces designed for other software to address, such as an API, listener, or message queue. Add those interfaces into your diagram as well.

1. http://c2.com/cgi/wiki?GalacticModelingLanguage

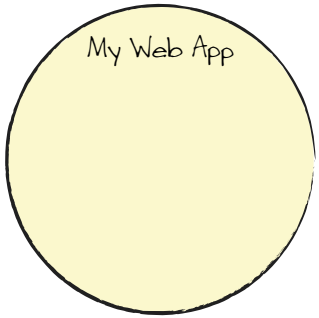


Figure 5— Starting a context diagram

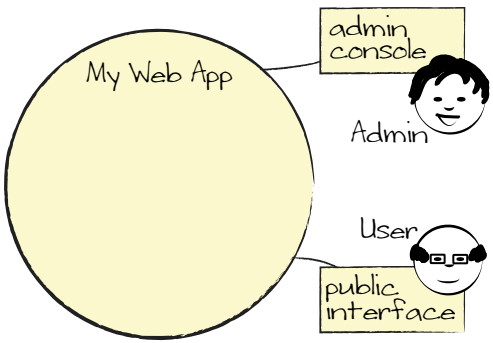


Figure 6— Adding the interfaces

Step 2: Map External Dependencies

Next add boxes representing all the other external systems or services that your software connects to. In our case we only had one external dependency: a payment gateway (Figure 7, *Context diagram for an example web application*, on page 90).

You may have many more external dependencies than this example. If you are working on a web-based application, you may have external services in addition to a payment gateway, such as a cloud-based media repository,

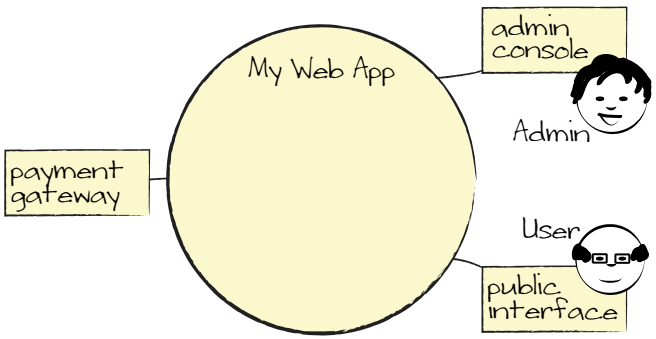


Figure 7— Context diagram for an example web application

external authentication services related to a single sign-on (SSO) feature, and external data sources, such as Twitter.

A complex enterprise application may connect to a dozen or more interrelated subsystems. Even a straightforward batch processing application that picks up data from one system, manipulates it, and passes it onto another will have two external dependencies: one for the source of the data and one for the receiving system.

Step 3: Fill in the Internals

Up until now the diagram has only represented the context in which your application, represented by the large empty circle, lives. Turning your attention to the inside of your system, it's time to add your system's component parts to the inner circle of your diagram.

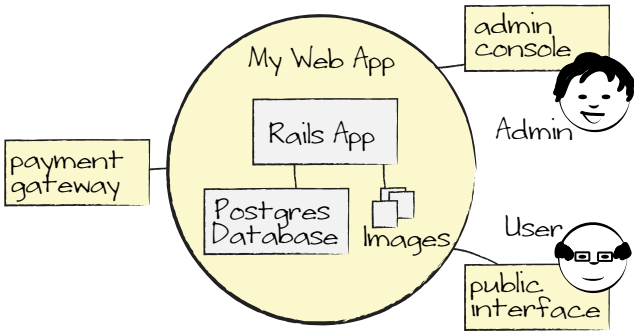
First consider where your code is deployed. If your team is working on an application that compiles into an executable binary that gets installed as a desktop application, this is quite straightforward. If your system is sufficiently large, there may be executables deployed across multiple servers.

The example in these diagrams is a web-based application built with Ruby on Rails. The ecosystem diagram could include fine-grained details like the Rails architecture, web server, and application server that Rails runs on. However, in this case adding those details will not help with exploring. I did not vary anything about those aspects of the system while exploring; it would

not have helped me discover the kinds of risks I was looking for. So I simplified all that into a single box labeled “Rails App.”

The next thing to identify is where your software stores data. It may be a database, files, or a combination of the two. When looking for where your software stores data, consider any kind of data. Your software may use files for storing configuration settings or preferences; those count. Also include log files and temporary files.

In the case of my example, we had both a relational database on the back end and images stored in files on the file system:



If you are working with a system you don’t know well and where there is very little documentation, you may need to use tools to help you uncover the parts of the system.

Note that whether to include or exclude elements of your system in your ecosystem diagram is a judgment call. If a given file is part of the overall system but completely outside the control of your software, it may not be worth including it on your diagram.

In this case, exploring things that can go wrong with web servers and application servers would not have helped move this particular project forward. So I did not include the Apache configuration file, httpd.conf, on my diagram. However, I did include the relational database that stored user data because exploring variations in the production system data was extremely relevant.

Avoid Analysis Paralysis

If you are dealing with a system that you did not have a hand in designing, it may seem an overwhelming task to figure out what all the dependencies are. In order to avoid suffering from analysis paralysis, try the following:

- When in doubt, start simple. You don't need to include every detail in order to have a useful diagram.
- Get others involved. You might not have a full picture of the ecosystem, but if you team up with others you'll have more perspectives to draw from.
- Share what you have. Even if you cannot get others to help you create the diagram, just sharing it may prompt others to offer corrections or additional details. So don't wait until you think you're done to share it.

If you find yourself obsessing over details, pull back and run an exploratory session with what you have so far. This will help you determine which details are relevant and which are not.

Finally, as the statistician George Box famously said, "All models are wrong; some are useful." Don't worry about whether or not your diagram is perfectly accurate; worry more about whether or not it gives you ideas to support your exploration.

9.2 Trust Boundaries

In *Object Design [Wir02]*, Rebecca Wirfs-Brock introduces the concept of a *trust boundary*, a place where your software connects to other software that it cannot assume will always play by the rules, such as another system, one that you don't control.

For example, when you look at your ecosystem diagram, consider the public interfaces. Those interfaces represent a trust boundary. On the other side of the interface is a system or a human you cannot control. Inside your implementation, there is code that protects the logic at the core of the system from having to worry about malformed inputs. The core is therefore inside the trust boundary.

The thing about trust boundaries is that it's not always obvious they exist. It's common to assume that core assets like the database are inside the trust boundary because there is code in the system that protects the database from getting bad data. However, if your system relies on a relational database, there are other ways to get data in and out of it than just through the official channels in the system.

In fact, on several client-server systems I worked on, the technical support representatives routinely walked customers through a process of fixing data

directly in the relational database. This is dangerous, of course. However, in cases where a bug in the code caused problems for the users, it was sometimes the only way to fix things.

Therefore, for the purposes of exploring, a trust boundary exists at every point in the system that you can control. Databases, file systems, and network connections are all examples of trust boundaries.

Once you identify a trust boundary, your task as an explorer is to violate that trust. Enter invalid data, corrupt files, or otherwise take actions that the system is not expecting. This leads us to the next technique for using your ecosystem diagram to explore the system: What If?

9.3 What If?

In the What If? game you use your ecosystem diagram as a map to suggest ways of breaking the integrity of the system.

The physicist Richard Feynman illustrated this technique perfectly in a story he told in *Surely You're Joking, Mr. Feynman! [Fey97]*.

While meeting with engineers designing a safety system for a nuclear production facility at Los Alamos, Feynman placed his finger at random on a symbol in the blueprints. "What happens if this valve gets stuck?" he asked. Feynman's intention was to use the engineers' reactions to help him figure out what the symbol meant. He expected them to say "That's not a valve, sir. That's a window." Instead, here's how Feynman described it:

> So one looks at the other and says, "Well, if that valve gets stuck…" and he goes up and down on the blueprint, up and down, the other guy goes up and down, back and forth, back and forth, and they both look at each other. They turn around to me and they open their mouths like astonished fish and say, "You're absolutely right, sir."

In the What If? game, you walk through your diagram of the ecosystem and ask questions like these:

- What if this connection goes down?
- What if the service at the other end of this connection is not responding?
- What if the network is so slow that the request times out before the response comes back?
- What if this file is missing?

The difference between your situation and Richard Feynman's is that you don't have to restrict your questioning to thought experiments. You can begin

to answer these questions with empirical data. Try disconnecting the network, halting servers, and removing files. Watch for indications that a single point of failure could bring down the entire system.

Here are some specific ways you can discover how your software fares when things go horribly wrong.

Network Connections

Wherever you have a network connection in your ecosystem diagram, try these methods:

- Disconnecting it
- Putting part of the system on the other side of a firewall or network segment
- Slowing it down

Years ago, testing around network conditions like this was difficult. If I wanted to slow down a network, for example, I would insert a modem between point A and point B. The alternative was very expensive network simulation tools. These days you can use tools like `netem` (included with Linux) to simulate network conditions like low bandwidth or high latency.

Files

Wherever you have a file in your diagram, try the What If? game with these methods:

- Deleting the file
- Corrupting the contents of the file
- Replacing the file with an empty one
- Making the file huge
- Configuring the software to read/write the file from a remote drive somewhere else on the network
- Making the file read-only
- Removing all permissions from the file
- Locking the file by opening it with another process
- Filling up the hard drive so there's no room for the file to expand

This list is just a start. You could exercise the software in many other ways to discover how well it handles file-related error conditions.

Unless it was explicitly designed to handle error conditions like these, software can behave very badly when it encounters problems with files. All too often,

the software is designed as though the file lives inside a trust boundary. However, users will do all kinds of things to files that we cannot anticipate.

Back in the early 1990s I was working for an internal IT organization supporting users on a system where the menus that users used to navigate through the system were all defined in plain text files.

One day I got a frantic call from a user in the accounting department. "YOU BROKE IT," she yelled at me. Baffled, I trotted down to her work area to find out what the problem was.

Once I saw her screen, I understood the problem. The menu options she used to do her job were all missing. She was panicking because she couldn't do her work.

After a little investigation, I found that the text files that contained all the menu commands were completely corrupted. The files had been in plain text, but now they were in binary. When I asked the user, in as diplomatic a way as I could muster, whether she had perhaps opened and subsequently saved the files in the word processor, she grudgingly admitted that she had.

When we had deployed the system we simply hadn't considered the possibility that users would become curious, poke around in the file system, and open the menu files. We didn't control the software that used the plain text menu files, so we couldn't fix the problem with code. However, we could (and should) have locked down the files with read-only permissions so that users could not shoot themselves in the foot.

The key lesson is that even if it seems like a file is on the inside of a trust boundary, it might very well not be. Experiments like the ones listed above can help you characterize the risks of unforeseen situations.

9.4 Follow the Data

You learned about following data in Chapter 7, *Explore Entities and Their Relationships*, on page 67. You can apply that heuristic even better by using your ecosystem diagram. For any given piece of data, trace through the connections in your ecosystem diagram to see all the places data might show up.

Consider Figure 7, *Context diagram for an example web application*, on page 90. If a user enters data through the user interface, that data goes into the database and could then show up in the administration interface. That data may also go to the payment gateway and could show up in the log file. These are all places to look for symptoms of problems due to invalid or potentially harmful data. In exploring the system, you would want to watch for possible

vulnerabilities to SQL or JavaScript injection attacks in each of these areas, not just in the user interface.

9.5 Practice Sessions

Try diagramming the ecosystem of your system. Identify user interfaces, programmatic interfaces, other systems that integrate with your software, and external dependencies. Map the internal parts of the system, including components and data storage mechanisms. As you're diagramming your system, identify the boundaries that separate things that your system controls from things it doesn't.

When you have your diagram close to done, run one more exploratory session with the What If? game. What if this network connection is severed? What if that file is locked?

As you explore, follow the data all the way through the system. Discover side effects and risks related to invalid data entering the system from any of the interfaces.

Part III

Putting It in Context

CHAPTER 10

Explore When There Is No User Interface

It is easy to see how the techniques for exploring a system apply when there is a user interface to manipulate. You can see and manipulate fields and controls. Sometimes it is a little more difficult to see how to apply these techniques on systems that don't have a GUI: servers, APIs, embedded systems, and batch programs.

The techniques in this book apply to absolutely any kind of software.

None of the analysis techniques require a user interface to make things visible. Consider the most basic analysis technique: identifying things to vary. You can always find interesting things to vary even if they are not exposed in a user interface. Imagine a batch processing program: it reads data from a file, transforms it in some way, and sends it on to another part of the system. You could vary the data in the file, the size of the file, or the count of records in the file.

Further, heuristics like Zero, One, Many are abstract building blocks that work regardless of the interface you use to manipulate the system. Consider that hypothetical batch processing system again. You can load in a file with zero, one, or many records. You can violate the domain-specific rules for the values in fields in the file. You can leave fields blank. Indeed, I have seen bugs triggered by each of those conditions in real-world batch processing systems: programs that crashed if the file was empty, if any of the fields were empty, or if any fields contained unexpected data.

Any system has at least one kind of interface, and usually more. The interface isn't always for humans. Sometimes it's to connect with other systems. It's still an interface, and that means you can still use it to explore.

In this chapter you'll see four examples of exploring non-GUI interfaces: a Java API, a JavaScript function, a web service using XML, and a server.

10.1 Exploring an API

I had just completed a session on exploratory testing at an internal conference for a company. One of the programmers in the audience approached me after the session. Let's call him "Colin."

"So," he began as he approached me, "this exploratory stuff sounds good, but it doesn't apply to what I do. I write low-level code, APIs that other programmers use. It looks to me like exploratory testing is only relevant to GUI-level stuff."

I immediately regretted doing my demo with a web application, and I knew that no amount of explaining now would erase the image of a JavaScript-centric, graphically rich interface from his mind.

"I don't think that's true," I replied. "How about if we pair on something you're working on right now and see if the heuristics I just went over could apply to your code?"

Colin agreed and led me over to a computer.

"Here's a little side project I'm working on," he said as he brought up his editor. "It's a library with functions for processing text." He pointed to a particular section of code. "This is a function that can compare two strings and score their similarity. It implements a scoring algorithm derived from cosine similarity and returns a value between 0 and 1,[1] where 0 means the two strings are not at all similar and 1 means the strings are a 100 percent match."

My mind automatically started analyzing the variables. I looked at the signature of the function we would be exploring. It took two, and only two, string parameters:

```
public double calculateSimilarity(String stringA, String stringB)
```

Because Java is a strongly typed language, the Java compiler would enforce the type and number of parameters. That meant that I could not vary the type of data I used in calling the function. For example, calling it with integers would be an invalid test:

```
calculateSimilarity(3, 5); // invalid because integers aren't strings
```

And I could not explore it by varying the number of parameters I passed in, so the Zero, One, Many heuristic would not apply to the number of parameters:

```
calculateSimilarity();    // invalid because it requires two parameters
```

1. http://en.wikipedia.org/wiki/Cosine_similarity

More accurately, I could vary these things, but then I would be exploring the Java compiler and not the function we had set out to explore. So I mentally discarded these tactics and considered what I could vary with two strings.

The length of each of the strings was a variable, as were the characters in the strings. The similarity and difference between the lengths and characters represented even more variables. Finally, the output score was a variable. I decided to focus on the output first and suggested that we find ways to get scores of 0 and 1.

Together, Colin and I wrote a little program that called the function and returned the result. We tried values that we were sure would be a perfect match, resulting in a score of 1:

```
calculateSimilarity("a", "a");
```

The function returned a 1 as expected. Next we changed our program so that it called the function with two completely different values.

```
calculateSimilarity("a", "z");
```

The function returned 0, again as expected. Colin was starting to look a little bored; these tests were too simplistic. He didn't realize I was just starting. I modified the program to call the function with two zero-length strings.

```
calculateSimilarity("", "");
```

The function returned 1, a complete match. So I tried with null values:

```
calculateSimilarity(null, null);
```

The program threw a Java exception and exited. Colin made a note.

Next I experimented with the length of the input strings. I typed a paragraph worth of nonsense and assigned it to a variable:

```
String myString = "a very long paragraph ...
calculateSimilarity(myString, myString);
```

(My actual paragraph was about a hundred words long.)

The program ultimately returned the expected value of 1, indicating that the two strings were an exact match (as they should be since they were exactly the same string). However, I noticed that the program paused significantly before returning the results.

So I decided to try comparing two very long strings. I opened a browser to Gutenberg.org and copied the first chapter from Mark Twain's classic book

Tom Sawyer.[2] I pasted the value into a variable and compared it against itself again. We waited. Nothing seemed to be happening. I opened a system monitor. The computer's CPU was pegged at 100 percent. Something was happening; we just couldn't see what.

Colin took another note.

"Looks like there's a performance issue," I observed.

Colin nodded. "And it would be a huge problem if I wanted to use this to detect plagiarism," he said. "Then I'd be comparing a student's paper against multiple sources, and it would just take too long to process." I glanced over. He looked thoughtful.

"We could kill the process and try some more experiments," I suggested. "I'd like to see how it handles other kinds of text, like accented characters, Japanese or Chinese characters, nonprinting characters, and different kinds of whitespace."

Colin laughed. "I don't think you need to. You convinced me," he said. "Exploratory testing is most definitely not just about GUIs. I'll explore more later. Let's go back to the conference now."

10.2 Exploring a Programming Language

As a programmer, it's important to understand the ins and outs of whatever language and libraries you use. Misunderstanding how the underlying technology behaves is a sure recipe for creating bugs.

In his comically genius "WAT" video,[3] Gary Bernhardt demonstrates how illogical JavaScript and Ruby can be. It's a fantastic example of exploring programming languages to discover quirks and surprises that can bite the unwary programmer.

Inspired by the video, I decided to explore the JavaScript sort() function. I began with a simple case:

```
["b", "c", "a"].sort()
```

This returned the results I expected:

```
a,b,c
```

So far so good. I began brainstorming things I could vary: the count of items in the array (Zero, One, Many), characters in strings (contents and character sets), and the types of objects (strings, numbers, objects, arrays).

2. http://www.gutenberg.org/ is an excellent source of long passages of text.
3. https://www.destroyallsoftware.com/talks/wat/

First I tried varying the count of items to be sorted. Here are the conditions I tried and the corresponding results:

```
> [].sort()


> ["a"].sort()
a

> Array(99999).sort()
,,,,,,,,,,, // ... and lots more commas
```

There was nothing particularly interesting there. So I turned my attention to data types. I decided to try numbers:

```
> [7, 3, 11].sort()
11,3,7
```

Whoa! JavaScript wasn't sorting numbers as numbers. It sorted the numbers as characters. 11 is numerically greater than 7 but alphabetically before 3. A quick search on the Internet revealed that if you want something other than an alphabetical sort, you have to pass in a function, like so:

```
> [7, 3, 11].sort(function(a,b){return a-b})
3,7,11
```

This is exactly the kind of surprise I was looking for, a language quirk that can lead to surprisingly bad behavior if you don't know about it. Worse, because alphabetic and numerical sorts only yield different results if the numbers have a different number of digits, it's the kind of subtle thing that's hard to pin down.

Moving on, I decided to experiment more with data types. Arrays in JavaScript can hold any type and can even mix types. So it's perfectly valid to have an array that contains both numbers and letters. So I tried this:

```
> [1.1, 0, "a"].sort()
0,1.1,a
```

I was back to things that were not all that interesting. What about arrays, objects, and special values?

```
> ["a", {"foo": "bar"}, Infinity].sort()
Infinity,[object Object],a
```

Now things were getting interesting again. Infinity was smaller than an object? A little more experimenting showed that once again the results were coming out in alphabetic order. *I* comes before *O* in the alphabet, so Infinity is smaller than Object. What if I passed in the numeric compare function again?

```
> ["a", {"foo": "bar"}, Infinity].sort(function(a,b){return a-b})
a,[object Object],Infinity
```

The key lesson learned here is that no matter what kind of things you have in an array in JavaScript, the default behavior is to cast them to a string and then sort alphabetically. If you want different behavior, you have to override the sort function.

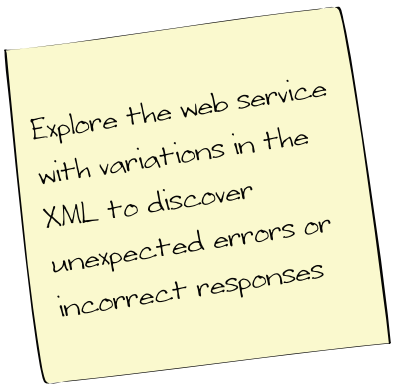
The other thing that is important to take away from this example is that exploration isn't just for production code. Exploratory techniques can help you to characterize the capabilities and limitations of the technology on which the code is built.

10.3 Exploring a Web Service

By now you have probably noticed a pattern: no matter what kind of thing you are exploring, no matter what kind of interface it has, you can always identify interesting things to vary. Let's look at one more example. Some years ago I was exploring a web service. The inputs to the web service were all in XML, like this:

```
<searchFilters>
   <filterBy>
    <fieldName>category</fieldName>
    <value>vacuums</value>
   </filterBy>
</searchFilters>
```

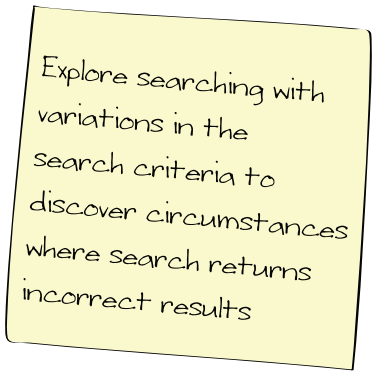
My charter was this:



I began the session by analyzing variables and quickly brainstormed the following variations:

- Count of filterBy nodes: Zero, One, Many
- String variations for fieldName and value, including empty, contains unicode characters, contains SQL injection attacks, and so on.
- Whether or not the content in the filterBy node matched actual available field names
- Whether or not the XML was valid
- The expected number of records found by the search (Zero, One, Many)
- Whether or not searches were case sensitive

At that point I stopped. My list was already quite long, and I hadn't even done any exploring yet. I decided that the scope of my session had become too big, so I peeled off the searching tests for a later charter, making a note to myself:



I then executed the most basic test: filter the data on a simple condition that I could easily replicate in the UI to check the results. This demonstrated that searching basically worked, so now I could explore more deeply.

I tried a query with no filterBy nodes:

```
<searchFilters>
</searchFilters>
```

I tried a query with no filters inside the filterBy node:

```
<searchFilters>
   <filterBy>
   </filterBy>
</searchFilters>
```

I tried a query with an ampersand in the category name:

```
<searchFilters>
   <filterBy>
    <fieldName>category</fieldName>
    <value>cleaning & supplies</value>
   </filterBy>
</searchFilters>
```

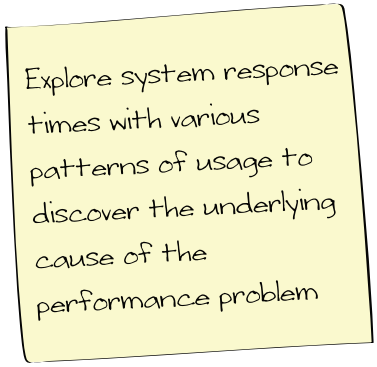
And so on. With each experiment, I learned a little more about how the web service behaved. I discovered that the system provided the correct error message on malformed XML, that it was not vulnerable to basic SQL injection attacks, but that it did not handle special characters like & correctly unless the characters were UTF-8 URL encoded. (So, for example, the & character needed to be sent as %26.) As my understanding grew, I expanded my exploration to include more variations on the input text as well as combinations of variations of inputs in the fieldName and value nodes.

10.4 Characterizing Bugs

Sometimes you explore to find problems. Other times you explore because you already know there's a problem and you need to characterize it in more depth. The techniques in this book work for both situations.

Consider, for example, the case of a web service that seemed to get slower and slower the longer the server had been up. Various team members noted that if the server slowed to a crawl, restarting the server process made the performance problems go away. So one of the programmers on the team designed a series of experiments. He did not think of it as exploratory testing, but that's exactly what he was doing.

The programmer even had a charter, although he would not have called it that:


Explore system response times with various patterns of usage to discover the underlying cause of the performance problem

The programmer wrote a little script that hit the server repeatedly with GET and POST requests. Then he ran his experiments in a series of time-boxed sessions, tweaking a handful of variables each time. He ran the script for shorter and longer durations. He turned caching on and off on the server side. He increased the amount of data being sent to the server.

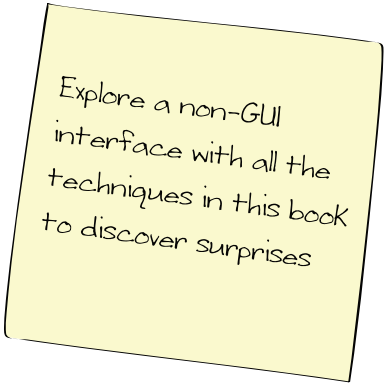
Ultimately the programmer discovered that increasing the load on the server sometimes triggered a time-out in a lower-level library within the server-side code. When the server started timing out, its responses slowed.

Just as with the previous example, notice that there is no user interface involved. The programmer's script issued GET and POST requests to an HTTP server. Yet the programmer considered all the same kinds of analysis techniques and heuristics that could apply for a GUI: he designed his experiments to vary the number, size, frequency, and timing of the requests.

Just as it's all too easy to dismiss exploratory testing as something particularly suited for GUIs, it's also tempting to think it's an activity that's performed exclusively by testers. As you have seen in this chapter, neither is true. The exploratory techniques in this book apply at all levels, from full stack system testing down to exploring the behavior of individual functions in code. Any project team member can apply these techniques.

In the next chapter we'll look at even more ways to apply these techniques to characterize the behavior of existing systems.

10.5 Practice Sessions



Revisit the context diagram you drew in Chapter 9, *Explore the Ecosystem*, on page 87. Notice all the interfaces where you can interact with your system. Find non-GUI interfaces: APIs, batch processors, and servers. Choose one of those interfaces and then use all the techniques in this book to explore it.

CHAPTER 11

Explore an Existing System

Ideally, you start exploring software while it is still under development, the second there's something executable. Exploring early and often throughout the development cycle means that the team can address issues while they're still small, and even prevent entire categories of problems.

Unfortunately, that's not always possible. Sometimes you're dealing with an existing system. All too often in such cases the software is an unknown quantity. It's in production, it fills a business need, but no one in the current organization quite understands exactly what it does or how it works. The original team members have long since gone on to other projects or even other companies. The documentation is woefully out-of-date. Even the most experienced subject matter expert available only understands a small piece of the puzzle.

In this chapter, you'll discover a structured approach for applying exploratory techniques to discover the capabilities and limitations of an existing system. You'll discover what a recon session is and how it can help you get a leg up on understanding the system as well as how it can help you assess the scope of the effort needed to explore the system fully. You'll learn a simple approach to interviewing stakeholders to glean important insights into the current state of the system. You'll also discover how to use exploratory techniques to pinpoint difficult-to-nail-down intermittent defects, a condition that can plague legacy systems.

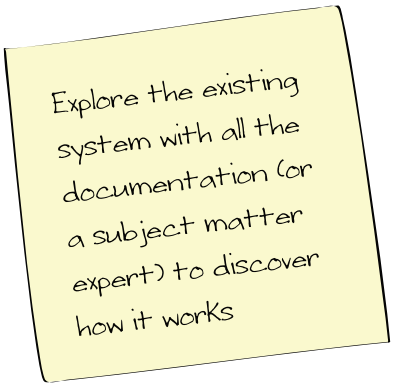
11.1 Starting with a Recon Session

The biggest difference between exploring features under development and using exploratory techniques to characterize an existing system is in the kind of information you are seeking. When you're exploring new development, you're usually looking for emerging risks and vulnerabilities resulting from

design and implementation decisions. When you're exploring an existing system, you may want to discover risks, but first you need to find out what it does, what it interfaces with, and how the pieces and parts all connect together.

A recon session is a special kind of session in which you begin mapping the territory of the system. At the end of a recon session you'll know more about the scope of exploration required and will have started to identify techniques to use in subsequent sessions. You'll also have at least some insight about how the parts of the system work together. You might even have an initial read on the overall quality level of the system.

A charter for a recon session is typically something like this:



The recon session sets the stage for all the other sessions and missions. As you execute this charter, you will learn about the following:

- *The ecosystem in which the software under test resides and the touchpoints you can use for visibility and control*: What does the software we're testing connect to? What system resources does it use? What public and private interfaces does the system provide where you can provoke, monitor, and verify system behavior? (See also Chapter 9, *Explore the Ecosystem*, on page 87.)
- *Variables*: The things we can change or cause to be changed. Some variables are obvious, like system inputs and outputs or configuration settings. Others are much more subtle, such as sequences of actions, timing, and attributes of data in the system. (See also Chapter 4, *Find Interesting Variations*, on page 33.)
- *Obvious vulnerabilities and potential risks*: These will begin to suggest the charters you'll want to set for future sessions.

The Easy Button Video

James and Jon Bach coined the term *recon session* to describe the first few sessions they would do when investigating a new product. They made a short (4-minute) video showing a recon session they conducted on the Staples Easy Button: a device with just a single button.[a]

The subtitles in the video name each testing technique: over a dozen in total. Some important techniques they identified in the video include the following:

- *Claims testing*: comparing the actual behavior of the system with the claims made on the packaging or marketing literature
- *Documentation review*: using the documentation to understand the expected normal usage of the system
- *Product analysis*: examining the system to understand its component parts, how it is put together, and the mechanisms available for interacting with it
- *Realistic scenario testing*: exercising the system in a way that is consistent with realistic usage scenarios, including realistic stress conditions
- *Stress testing*: approaches for stressing or abusing the system, including James Bach's infamous "shoe test," any test that is consistent with putting a shoe on a keyboard

Even when exploring a simple device with just a single button, James and Jon demonstrate how a thorough investigation necessitates taking a variety of approaches to gain multiple perspectives on a target system.

a. http://www.youtube.com/watch?v=Vy0I2SB5OLo

By the end of the first recon session, you should have a rough map of the ecosystem, a list of the interesting things to try varying in subsequent sessions, and a list of possible charters. However, you may want to conduct more recon sessions if you find it difficult to answer these questions:

- What does it do?
- How does it take input?
- How (or where) does it produce output?
- What is a simple input or sequence of actions, and what is the corresponding output or result?
- How does the environment or configuration affect the output?
- Are there any ways to interact with the system that skirt around the intended public interfaces? (For example, are there hidden configuration

files you can manipulate? Is there data stored in a location you can access directly? Are there individual components in the system you can access directly, bypassing the more commonly used public interface?)

- How can you trigger error conditions?
- What kinds of things are likely to happen if you use the system in a way that is not intended, or intentionally trigger error conditions?

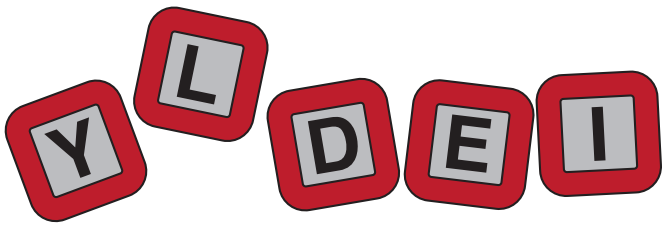
Once you can answer these kinds of questions easily, you're ready to move on to sessions focused on specific aspects of the system. Use the guidelines in Chapter 2, *Charter Your Explorations*, on page 11, to identify charters for those subsequent sessions.

11.2 Sharing Observations

If you want to learn about an existing system quickly, it's best to work with other explorers and discuss your findings. This makes intuitive sense: more people can cover the same ground more quickly. Each person brings a unique perspective and skills to play and thus will notice things that others won't.

However, it's not just a matter of many hands making light work. Working with others can accelerate the pace of discovery. A group of people exploring collaboratively can share information. As a group, you will discover more in the same amount of time by working together than if you all explored different aspects of the system individually. By pooling your insights, you make more effective use of your time. I have seen this many times in the real world, but the power of sharing information is particularly noticeable in my testing classes.

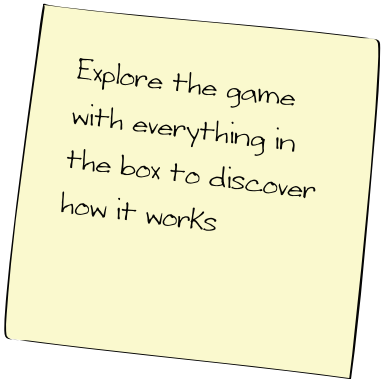
When I teach exploratory testing classes I ask participants to explore a hand-held electronic game. My current pick is Scrabble® Flash, a pocket-size electronic word game where players have five electronic tiles, each of which displays a letter. When players assemble the displayed letters into words, the game increments the players' scores.



The great thing about this game is that it's simple, yet it has a near infinite number of interactions to explore. The tiles behave differently when they're

next to each other than when they're separated. Each tile has a state, and the game as a whole has a state. As you might imagine these states can interact in interesting ways. There are any number of other variables as well: the position of the tiles relative to each other, the angle of the tiles, the distance the tiles are apart, the current letter displayed, the current score, the possible high score for the game, and so on. The fact that the game is housed in a black box just makes it even more appropriate: it's an exercise in true black-box testing.

We do a recon session for the first exercise of the day. I ask participants to do the following:



It's interesting to watch different approaches. Some people start with the documentation: they spend the first several minutes of the session puzzling through the densely written instructions printed on tissue-thin paper. Others ignore the instructions completely and start playing with the devices.

At the end of fifteen minutes, I ask everyone to pause and call for a debrief. Most people report very similar findings: the tiles seem to communicate; the tiles appear to be identical; there are three distinct games. Inevitably some people will report on surprises that not everyone else noticed: the tiles are so identical that they can be swapped between sets; some of the games can be played with just four instead of five tiles; and some of the words the game recognizes are a little weird. (Rin? Rin. It's in the Scrabble® dictionary.)

Occasionally some people will dig even deeper in those first fifteen minutes: they'll discover subtleties about how the game functions when varying tile orientation and distance, they'll discover the reset button on the back of the tiles, or they'll experiment with timing and interruptions.

As each person offers some new piece of information, others in the room pick up on the clues and experiment with their devices to see for themselves.

Within half an hour, the room as a whole has a solid understanding of the basic mechanics that make the game work and is ready to dig into much more sophisticated exploration. The shared learning is what makes this possible. If I left people to explore on their own, it would take hours for each person to discover all that information independently.

11.3 Interviewing Stakeholders to Gather Questions

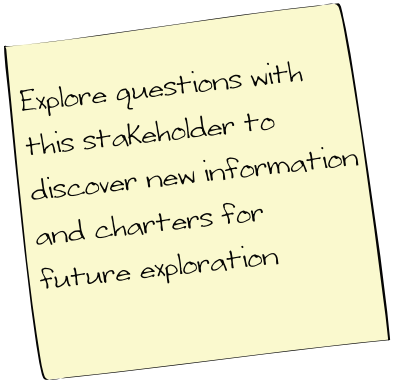
Once you are sufficiently fluent in your understanding of how your target system works, you're ready to interview stakeholders to understand what questions they have about the system. You are in a position to start answering their questions, and their questions will help you explore more effectively.

Here's a list of possible stakeholders:

- The person or people who charged you with exploring the target system
- Technical support people who have to respond to user or customer issues
- Programmers who have to maintain the system
- Business users who rely on the system
- Product managers or analysts who have to draft new requirements for changes to the system
- Anyone else who has a stake in how well the system functions

You don't have to set aside a huge block of time to talk with these people; you can query them in a quick, informal chat and then schedule more time with them later if it seems that more time would be mutually beneficial.

It might help to think of your stakeholder interviews as another kind of exploration:



To that end, you want to craft questions that convey the context and something about what you have already discovered while simultaneously asking

open-ended questions that invite your stakeholders to share anything they think is relevant. This takes practice.

Consider a hypothetical example. Imagine you are exploring a legacy billing system. It's a mission-critical system (no billings leads to no cash flow, obviously a bad situation). The system needs to be updated to reflect ongoing changes in how the company does business, but no one actually understands how the billing system works. So making the needed changes will be dangerous and scary. Your task is to characterize the existing behavior well enough to reduce the risk of unintended consequences from the upcoming changes.

You head down to see a stakeholder, the head of the customer service department. Her team's focus is not on billing, but because of the way your company does customer service, the department members have to have access to billing and payment records to do their jobs. Consider which conversation opener is likely to lead to a more fruitful conversation:

> So, what should I know about how your group uses the billing system?

Or…

> I'm exploring the billing system to understand how it interoperates with other key business systems, including your customer management system. I've discovered that orders, invoices, and payment history are linked between the two systems. Can you tell me more about how your group interacts with the billing system, and in particular, with customer orders, invoices, and their payment history?

The first question is so open-ended that it's difficult to even know how to begin answering it. The second question does a better job of framing the discussion. Notice that it accomplishes these goals:

1. It sets the context. Why are you here? What are you doing that prompts this conversation at all?
2. It gives a snapshot view of what you know already. This shows that you've done your homework and also gives the other person a cue about the level of information you're ready to absorb.
3. It asks a specific yet open-ended question that can take the conversation just about anywhere it needs to go.

As the conversation progresses, ask other people questions designed to invite them to share their understanding of the system, their opinions about it, their frustrations and/or joys in interacting with it, their hopes and concerns, their perception about overall quality, and the puzzles they've encountered.

Throughout the conversation, make note of capabilities that the other person mentions that you have not yet explored (or perhaps didn't even know about) or questions that your exploration can answer.

11.4 Selecting Techniques

After a couple of recon sessions and discussions with stakeholders, you should have at least a high-level view of the territory. This will enable you to see where to apply the various analysis and test design techniques to ferret out deeper insights. The challenge is in choosing which combinations of techniques are most appropriate to support you in your investigations and what questions you need to answer.

If you notice that the system seems to be sensitive to timing, see Chapter 8, *Discover States and Transitions*, on page 75.

If you see opportunities to vary the order in which you perform operations or if the system is user-facing, see Chapter 6, *Vary Sequences and Interactions*, on page 59.

If you notice that the system is heavily data-dependent, see Chapter 7, *Explore Entities and Their Relationships*, on page 67.

If you notice that the system interoperates with other systems extensively, see Chapter 9, *Explore the Ecosystem*, on page 87.

Along the way you will use the same kinds of heuristics that you would use for any other kind of exploration: Zero, One, Many; Some, None, All; First, Middle, Last; and so on. However, rather than looking for deep bugs, you're seeking to understand the system well enough that you can start to predict the effect your actions will have.

11.5 Recording Your Findings

Given that you find yourself having to characterize a legacy system because the documentation was inadequate, you'll do your future self and others a favor if you capture what you learn for posterity. You could do this in a formal way, but a lighter-weight approach can serve just as well. Here are some tips for capturing your observations with minimal additional effort.

While debriefing others about your findings, sketch key models (such as a map of the ecosystem or a state model) on a whiteboard. When you're done, take a picture of the sketch and post it to a shared space, such as the team's wiki. Write a few words of explanation to go along with the picture. It will take you half an hour or less to publish key insights about the system. You can always polish those snippets into a more formal document later if needed.

If the system involves a set of rules that determine a given output based on a set of conditions or inputs, capture a representative set of examples that illustrate those rules in a table. Better yet, if at all possible, use a functional test automation framework (such as Fitnesse, Cucumber, or Robot Framework) that supports expressing tests in a tabular format to automate the execution of those examples. Automating the tests may not be easy, but if the same sequence of actions can apply to multiple tests, they'll pay for themselves quickly.

Use your newfound expertise to help you choose a short set of repeatable smoke or regression tests that represent the core responsibilities of the system. Capture them in a lightweight way. (That is, rather than documenting step-by-step instructions, capture the essence of each test at a high level.) You can then use this list of tests as a starting point for automating regression tests or for a checklist of things to verify after code updates.

11.6 What About the Dreaded Unreproducible Bugs?

Cannot Reproduce.

It's the designation you dread on bug reports. Something went wrong. Once. Somewhere. And now no one seems to be able to make it go wrong again.

Perhaps you were the one who spotted the bug, and now you can't figure out what you did to make it happen. Or perhaps it's a report from a team member, another department, or worse, a customer. Whoever spotted the pernicious critter crawling through the code, you can't figure out what's happening. Yet you don't dare act as though the problem never happened. Just because you cannot spot the bug now doesn't mean it won't bite again.

The key to pinning down these mysterious intermittent defects lies in recognizing that any defect is absolutely 100 percent reproducible, but only if you can discover and control all the variables that affected the behavior.

I once worked on a utility program that ran on PCs. It was supposed to run quietly in the background, monitoring the system and only announcing its presence in order to alert users to imminent problems. Like a virus checker, the utility program relied on a hosted server to provide it with updated information about problems to monitor.

The intent of the program was to help prevent problems for users. Ironically, the program sometimes caused problems. Under some circumstances that we couldn't quite pin down, the program would crash, throwing up an ugly and scary error. The entire technical team tried and failed to understand what could possibly be causing the intermittent crash. There was no correlation

between the software crash and the user's activity, nor was there any other pattern that we could see.

Eventually we gave up. We didn't have time to chase a digital will-o'-the-wisp; we had software to ship. After the software shipped, we discovered that the rare failure was actually commonplace in the wild. It seemed like every customer saw the crash. We went back to trying to understand what could possibly be happening. Four of the strongest team members huddled in a room for a week pouring over source code and brainstorming possible causes.

In the end, the team members discovered the underlying root cause: one part of the system was trying to acquire an exclusive lock on a local data store, while another part of the system already had the data store locked. This happened more often in the field than in our lab because of two key variables: the level of activity on the client machine and the quantity of updates available to download from the server.

Once the team pinpointed the problem, they illustrated it with a simple state analysis. In hindsight, the underlying cause of the bug seemed obvious, but until we did the analysis to identify the variables, we couldn't see it.

Although we understood the root cause of the crash, we still could not control the utility to the point where we could make it happen on command. We had discovered the variables; we could not control them. Fortunately, identifying the factors that contribute to the problem is usually sufficient to enable you to make it happen more often, if not on demand.

If you are similarly faced with an intermittent defect, here are the keys to using the techniques in this book to identify it.

Gather Evidence

Find whatever traces of evidence you can from past instances of the problem: log files, screen shots, input data, console errors, and so on. For tips on looking for subtle cues, see Chapter 3, *Observe the Details*, on page 25.

Having gathered all the data you can, start looking for patterns. Consider:

- Does the problem seem to happen most often at a particular time of day, day of the week, or day of the month?
- Does the problem seem to happen most often on a particular computer or configuration, or when a particular user is logged in?
- Can you see any correlation between the problem and the timing of actions or level of activity?

Brainstorm Contributing Variables

Start listing all the variables you think might be affecting the behavior. Review Chapter 4, *Find Interesting Variations*, on page 33, for ideas about the kinds of variables that might be interesting.

Note that truly intermittent defects usually involve multiple contributing causes; it's almost never a simple matter of identifying a single variable. Even in my example of the crashing utility, where the answer was quite simple once we understood what was going on, there were two key variables at play: the level of activity on the client and the amount of updates available on the server.

Use State Models to Understand Timing

Most intermittent defects seem to be related to timing in one way or another. They show up when the conditions are just right, and hitting those conditions involves timing your actions just so. State diagrams, as described in Chapter 8, *Discover States and Transitions*, on page 75, are an excellent tool for understanding subtleties in timing. In particular, watch for transitory states: conditions that exist for a brief window of time and are not something you can control directly.

Collaborate

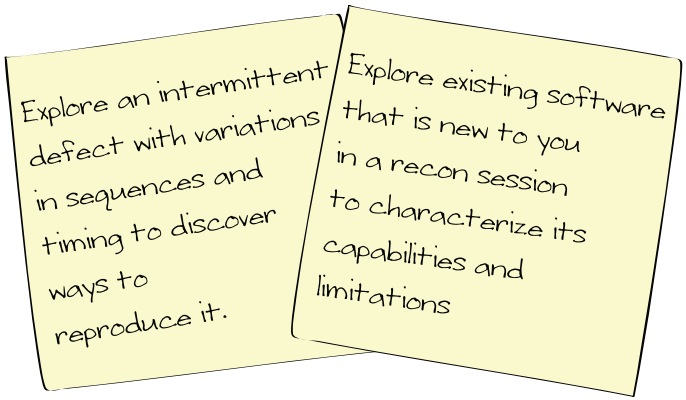
As with exploring the hand-held game in a classroom, a team of people collaborating and sharing information is much more likely to stumble on the root cause of the problem than a single person acting alone. Collaboration works particularly well if you are able to work with people who have a wide range of skills and areas of expertise.

The group of four people who ultimately identified the underlying root cause of the intermittent crash in the utility program included two senior-level test engineers with a deep understanding of the system as a whole and two senior-level programmers with a deep understanding of the data storage and threading model that the utility used. Prior attempts at pinpointing the problem involved individuals working in isolation. It wasn't until we had all the relevant skills at the table that the team was able to fully characterize the issue.

11.7 Practice Sessions

You could practice the ideas in this chapter with the software you work with every day or with software you haven't used before.

If you have any lingering intermittent defects, try using the ideas in this book to reproduce the issue. Identify variables that might contribute to the problem, especially subtle or hidden variables, configurations, sequences, and timing.


Explore an intermittent defect with variations in sequences and timing to discover ways to reproduce it.
Explore existing software that is new to you in a recon session to characterize its capabilities and limitations

Also try the techniques in this chapter on something that is new to you. It might be a new game, an open source utility, or even a web application. It could be anything, made by anyone. This is even more fun if you do it as a group activity, so you might recruit some friends or coworkers to do this exercise with you.

Start with a recon session. Map out the touch points. Find variables. Discover the capabilities and limitations of the software. See if you can figure out how it works or identify patterns in its design that suggest experiments to try. ("Aha!" you might think. "This application is sending data back to the server with every keystroke and mouse click. I wonder what happens if I speed up my interaction or slow down the Internet connection. Now where did I put that old dial-up modem?")

Once you have enough of an understanding about how the software works, identify a more specific charter and do another session. You might explore the software to discover catastrophic failures: make it crash, corrupt data, or do something it shouldn't ever do. On the other hand you might explore it to discover whether you can use it to accomplish some goal or for a particular purpose.

As you explore, use all the ideas in this chapter. Find ways to monitor and control the software, and vary the variables you identified using ideas from Appendix 2, *Test Heuristics Cheat Sheet*, on page 151.

CHAPTER 12

Explore Requirements

I once met a tester who didn't see how exploratory testing skills contributed to requirements discussions. "Why should I bother going to requirements meetings?" he asked me. "I can't contribute anything. I could listen, or I could wait until they figure out the details and then read the specification. They invite me but I just don't see any reason to go. Those meetings are long and boring. I'd rather read the book than see the movie."

I knew this tester was good at predicting risks and at exploring. I suspected that his team would benefit from his participation in the meetings. So I answered his question with a question: "When you build your tests on the specifications, do you ever find inconsistencies or errors?"

He nodded.

"Do you ever have questions?"

He nodded.

"When you ask those questions, do the answers sometimes lead to the discovery of new requirements?"

He nodded.

I paused a moment. He looked thoughtful. I considered capping the discussion with a rhetorical question like, "Don't you think it would be better for that information to come up sooner rather than later?" Instead, I chose a response that I hoped would help him connect the questioning he obviously did well with his testing skills: "Huh. Sounds like testing," I said. I waited to hear his response.

His face brightened. "Oh!" he exclaimed. "I get it. I hadn't thought of it that way before. I am testing the requirements. If I did that in the meeting, it could prevent entire classes of problems."

Absolutely. Even before there is anything implemented and ready to explore, it's possible to explore ideas about the software. In this chapter you will discover how to apply exploratory techniques right at the very beginning, during requirements discussions. You'll learn how to get into those requirements discussions if you aren't already, and what to do once you're there. You'll learn how asking the same kinds of "What if?" questions from the very beginning enables you to think through implications of decisions and discover potential issues before they manifest into actual risks. You'll also discover how requirements discussions can help you discover the most valuable charters to explore as features are implemented.

12.1 Getting Into the Requirements Meeting

No matter what software development process your organization follows, it's almost a sure thing that it has requirements meetings. They might even be called "requirements meetings." Sometimes they're called "design meetings." If your organization practices a variation on Agile, they might be called "user story workshops." Whatever these meetings are called, they're where key stakeholders meet to establish a shared understanding about the features to be implemented.

If you are not already part of these meetings for the software you're working on, you need to be. You need to hear the conversation firsthand, and your team needs the exploratory mindset that you can bring to the table. Spending a few minutes exploring around the edges of an idea, questioning assumptions, and connecting new concepts with the existing implementation can save weeks of rework.

The Problem of Being Left Out

As an example, consider the following situation from my own experience.

Many years ago I worked at an organization where most people thought that requirements meetings were a waste of time. Instead of holding a meeting with all the various stakeholders, the product manager would write a requirements specification and then meet directly with one or two of the programmers to walk through the document. By keeping the meetings as small and informal as possible, the product manager reasoned, the meetings would be blissfully short and focused.

The meetings were short and focused for the people who attended. Unfortunately, the following months during which the entire team attempted to implement and test the feature involved a great number of clarification discussions, often around the same rehashed topic, as new people were brought

into the discussion for the first time. The result was a great deal of time wasted on rework.

The worst case of this was that the team had implemented a server-side feature twice already and were preparing for a third attempt.

The feature was related to measuring the success of a given content item served up by the server so the marketing team could compare the effectiveness of variations on a given message. This kind of experiment, where a server randomly selects one of two versions of the same message for a head-to-head comparison of their effectiveness, is often called an A/B test. It's something marketers often need.

In our case, the product manager who was specifying the behavior for the feature was to be the primary beneficiary of it.

The product manager and lead programmer had already met to discuss the feature and had arrived at an agreement about how it would behave. It was my job to test the feature. When I saw the technical specification, I was baffled. It did not seem to me that the feature as designed would do what I thought the product manager required.

Misunderstandings from One-on-One Discussions

I visited the product manager. (Let's call him "Jim.") "Hey, Jim," I started. "I'll be testing the server-side metrics feature. Can you tell me how you plan to use it?"

Jim was only too happy to tell me about all the experiments he hoped to run by varying content and measuring the different levels of response. He went on for a full fifteen minutes about the marketing experiments he hoped to run with the new feature.

I grew even more concerned. "I read the technical specification," I said. "It looks to me like it won't do any more for your ability to run the kind of A/B tests you're describing than the current functionality."

Jim scowled at me. "I met with the lead programmer yesterday about this," he said. "It will be fine. When can you have it tested? I really need it in production."

I tried unsuccessfully to get Jim to listen to my concerns or answer my questions. He brushed me off gruffly. He didn't have time to discuss it with me, he said. He trusted the lead programmer. (Let's call her "Carla.") Jim told me that I obviously just didn't understand the feature, so he suggested I work with Carla to make sure I knew how to test it.

I tried having a conversation with Carla about the feature. I told her about the experiments that Jim wanted to run using the new feature. She dismissed my concerns with an airy wave of her hand. "Oh, that Jim," she laughed. "He says he needs all this sophisticated stuff, but he really doesn't. I'm building him exactly what he actually needs."

I tried unsuccessfully to get Carla to listen to my concerns, but she would have none of it. She brushed me off, saying that she didn't have time to discuss it with me. "You have the specification," she said. "Just test that it does what's described in the spec."

At this point I was convinced that Jim and Carla were talking past each other. I tried calling a requirements meeting, but both Jim and Carla declined the invitation. They had already met without me, they said. It was my problem to get up to speed.

I was becoming desperate. I was sure that if we finished implementing and testing the feature, it would be just like the other two attempts that had already failed to deliver on the promised value. If we had to throw away yet another implementation of the same feature, it would make it nearly impossible for us to deliver the rest of the promised features. I decided to call a test review meeting. Tests and requirements aren't just two sides of the same coin; tests express requirements. I knew that if I could get Jim and Carla together in a room with me to discuss tests it would inevitably turn into a requirements meeting.

Fortunately, my gamble paid off. Jim and Carla took pity on me and agreed to meet with me and the other testers on the project to define the tests for the feature.

A Test Review Becomes a Requirements Review

In the meeting, I stood at the whiteboard. I explained that I wanted to design some tests based on simple use cases. "Jim, when you and I met you had some ideas for experiments you hope to run once the feature is live." I held up a whiteboard pen. "How about if you walk me through one of those experiments?"

Jim outlined the simplest A/B experiment he had described to me in our discussion. As he talked, I made notes on the whiteboard.

I turned to Carla. "So Carla," I began. "Can you help me figure out how to use the feature as it's currently designed to run this experiment?" Jim smiled in anticipation.

Carla shifted uncomfortably in her chair. She looked at Jim, then down at the ground. She shuffled her feet. Finally she said quietly, "You can't."

Jim's smile evaporated. He was silent for a moment; then he exploded. "You WHAT?!?" It simply had not dawned on him until we were finally all in the same room discussing the feature all at the same time that he and Carla had misunderstood each other so very badly. It took someone with an explorer's mindset to question assumptions and discover the very large gap between what Jim needed and what Carla planned to deliver.

If you are being left out of the requirements meetings, as I was, your best strategy is to infiltrate them. (Tip: Bring cookies. The other people involved are less likely to kick you out of the meeting if you come bearing chocolate.) If you can't figure out where the real requirements meetings are being held, you can hold a test review meeting to serve the same purpose. Just make sure to hold that meeting before too much work has been done on implementation.

12.2 In the Requirements Meeting

Once you wangle your way into the requirements discussion, you have the opportunity to help the team discover ambiguities, assumptions, and gaps. You can suss out the Nevers and Alwayses for your software (see Section 5.1, *Never and Always*, on page 48), and in doing so you can help make sure everyone involved has a clear vision of the inviolate rules of the system as well. Also, many of the same heuristics you use for exploring software implementations can help you frame specific questions about expected behavior using concrete examples of inputs or conditions.

In this section you'll discover how to apply your exploration skills to discover the core value that the software is intended to provide and the expectations for behavior related to specific conditions with "What if?" questions.

Discover Core Values

Software exists for a reason. Maybe it automates a business process. Maybe it drives custom hardware. Maybe it provides entertainment.

By understanding the why behind each requirement, the team can make sure that the implementation serves the real need behind the stated requirement. Further, you can tailor your exploration, focusing on the most crucial usage scenarios. You can choose better representative data, configurations, and sequences. You can ferret out the risks most likely to cause serious problems in the field.

The best place to discover the core values of the system is in requirements meetings. Ask your business stakeholders about the core value they hope that the software will provide to the business. Try to understand out how the software fits in with the overall business strategy. You might also ask the questions suggested in *Core Capabilities*, on page 48.

As you listen to what your business stakeholders say, take notes. Listen for core features or capabilities. These are things that you will want to explore around in depth: if nothing else works in the system, these things must perform solidly.

Also listen for "-ilities": quality criteria such as reliability, scalability, usability, accuracy, and so on. These descriptors tell you a great deal about what kinds of information will be most important to your stakeholders. If accuracy is crucial and your exploration reveals a set of conditions under which the software provides inaccurate results, that information is extremely important and valuable.

Ask "What If?"

The perfect time to start applying your test design skills is in the requirements discussion. If your organization is typical, the group's focus is likely to be on understanding what the software should do under ideal circumstances (the so-called *happy path*). It's important to understand the happy path, of course. However, it's also important to understand what the software is supposed to do when faced with less-than-ideal circumstances. You can use your exploratory mindset to push the envelope. Ask "What if?" questions, using your test design skills to imagine possible error conditions:

- What if users do something they're not supposed to do? For example, what should happen if users attempt to log in when they are already authenticated? What if they attempt to log in while authenticated but use the wrong password? Are they no longer authenticated? (For more ideas like this, see Chapter 6, *Vary Sequences and Interactions*, on page 59.)
- What if the software does not have access to a necessary resource? For example, what should the software do if it cannot read the configuration file on startup? What if the configuration file is missing, locked, or corrupted? (For more ideas like this, see Chapter 9, *Explore the Ecosystem*, on page 87.)
- What if the input data varies from what's expected? For example, what if there are more records than the maximum, or fewer than the minimum? Or what if the XML in the web request is invalid or missing a required

node? (For more ideas like this, see Chapter 4, *Find Interesting Variations*, on page 33.)

Note that you have to be able to generate possible test scenarios on the fly to explore effectively in a requirements discussion. You have to be able to take a sketched-out idea of an implementation and immediately begin listing possible test cases in your head. Then you have to choose the most interesting or relevant of the cases you imagine and formulate them as "What if?" scenarios. It takes practice to do this well.

When you are sufficiently fluent in designing tests on the fly, you'll reap rewards on two levels. First, just by asking about possible risks, you're applying your exploration skills to prevent bugs. Second, the answers to your questions will help you calibrate your internal sense of what the system should and should not be able to handle. This in turn will ensure that your charters are on target.

Calibrate Expectations

Ron Jeffries is a programmer and coach and one of the signatories of the Agile Manifesto. He is active on mail lists, where he dispenses sound advice. He also sometimes gets into debates. In one such debate with Michael Bolton (the tester, not the singer), Ron commented, "I wish you testers would stop making stuff up!"

Ron may have intended the comment tongue-in-cheek. At the time, however, I was stunned at the accusation and felt defensive. All the frustration I had built up over years of seeing deferred defects I had found bubbled to the surface. I was alone, sitting at my kitchen counter, at the time I read those words. No one was around to hear me, but I shouted my reaction to the empty room anyway: "WE DO NOT MAKE STUFF UP!"

Then I remembered one particular project I had been on. I was testing an HTML parser back in the late 1990s. HTML was relatively new, and reading an HTML file required writing a custom parser. The programmer who was writing the parser had been working on it for a few months before I was brought on board as a tester.

I began exploring the almost-completed parser with simple little HTML files containing simplistic tags: tables, paragraphs, bullet points, horizontal rules. I found a few bugs. The programmer fixed the bugs, and I moved on to more challenging input with nested tables and nested lists and images and JavaScript. More bugs, more fixes.

Every week the programmer, the development manager, and I would meet to discuss the bugs I was finding. Every week the programmer frowned at me. As the weeks went on, the programmer's frown became a grimace and then an outright dagger-stare. I stood my ground against the programmer's hostility, but I could not mistake his furor over my findings.

The project wore on. I explored in even greater depth. I started experimenting with variations on invalid HTML: unclosed tags, nonexistent tags, tags in illegal places, and invalid attributes. I found more bugs, some of them so bad that they caused severe data corruption. I considered these bugs showstoppers and was quite proud to have found them before we shipped.

In a meeting to discuss some of the more egregious bugs I'd found, the programmer glowered, arms crossed in a sullen rage. Finally, as we were reviewing a bug report for a particularly egregious corner case involving invalid HTML, the programmer exploded: "NO ONE EVER SAID THIS PARSER HAD TO HANDLE INVALID HTML!!"

At the time, I was outraged that the programmer had such low standards for quality. However, sitting alone in my kitchen and reading Ron Jeffries's words, I suddenly empathized with the programmer. In all those meetings, I thought we were discussing quality and bugs. The programmer saw it differently: I was adding requirements on what should have been a small, simple feature.

In short, from the programmer's perspective, I was making stuff up. Every one of the invalid HTML bug reports I filed represented a new requirement and thus scope creep. It was a good thing I found the bugs, but it would have been so much better to hash out the scope earlier.

If you don't calibrate expectations with the team early on, you're likely to end up in the same situation: arguing about the real scope of the project under the guise of discussing bugs when it's too late to change course. If the team decides to act on the information you find, the project could slip. If they don't, you wasted your time.

12.3 Charter During Requirements Discussions

While you are asking questions about what the software should and should not do, you can also make notes about charters to explore later. During the discussion, be alert for anything that would go into the "with" section of a charter (Figure 8, *The "with" section of the charter*, on page 129).

- Other areas or features in the system that will interact with the feature or area under discussion

Figure 8—The "with" section of the charter

- Data sets, such as production data snapshots or existing test data
- Configurations of particular importance, such as key customer configurations or planned deployment configurations
- Use cases or personas that are of particular importance

Also be alert for types of information that would go in the “to discover” part of the charter template:

Figure 9—The "discover" section of the charter

- Never and Always conditions
- Open-ended questions around quality criteria, such as “What would the performance impact be if…?”
- Stakeholder questions and concerns, such as “Will the new UI design for this feature flow smoothly with the other features?”

As you are making notes about charters to explore, it's a good idea to calibrate those charters with the other stakeholders. You'll gain a great deal of insight about the extent to which your stakeholders will value the information that your explorations will yield. In addition, you may prompt an even more in-depth conversation about the scope of the requirements. One way to check in with your stakeholders is to ask a very straightforward question:

> "I just captured that concern as the following charter. Do you agree it's something we should spend time exploring during implementation?"

Another approach is to pose the question in terms of the kinds of risks you imagine your exploration might expose:

> "If I found problems related to the interaction between this feature and the saved searches feature, would you want to know?"

Calibrating your charters by asking questions like this can save you from wasting time exploring corners of the software your stakeholders are not concerned about. It can also help increase the confidence of your stakeholders because they know you'll focus on exploring risks they actually care about. Think of calibrating your charters as another example of establishing shared understanding within the team: you're establishing a shared understanding around what kinds of risks are important.

12.4 Active Reading

Face-to-face, realtime conversations are the best way to convey information about requirements. However, sometimes the requirements decisions were made before you were around. If the decisions are captured in documents, you may find yourself gleaning information about requirements more from documentation than from interactive discussions. In that kind of situation, active reading techniques can help you extract expectations from the written word. In this section, you'll learn an approach to active reading that can help you use static requirements documents to discover charters worth exploring.

Question the Paper

Active reading means you're questioning the document as you're reading it. It's a little like having a conversation, but with the written word rather than with an actual person. Instead of listening, you read a little. Just as you would in the requirements discussion, you mull over the new information and formulate questions. Instead of asking the questions out loud, look for the answers in the document.

The "What if?" questioning approach works here too. When you find statements about what must be true in the system, consider what should happen if it's false. For example, if you find a statement about a boundary, like a maximum amount of data or number of users, try to find a corresponding statement that says what should happen if that boundary is violated.

Break It Down

Categorizing the information in the document can help you delve deeper and stay focused. One model that works particularly well for active reading of requirements documents is Inputs, Processing, Outputs. To use this model, take a piece of paper and label four areas: Inputs, Processing, Outputs, Questions. You'll make notes in each area about the associated topic.

Inputs are anything the software receives or takes in. For example, you might discover references in the document to user-supplied data, files generated by a batch job in another part of the system, or events that the system monitors.

Outputs are anything the software produces. They might include reports, messages to the user, messages on a console or in a log, or even events consumed by another part of the system.

Processing is anything that describes how the software turns the inputs into outputs. Any information you find on algorithms, actions the software takes behind the scenes, or references to interim data goes here.

Finally, make note of questions inspired by your search for information about inputs, processing, and outputs. Perhaps you uncover more "What if?" questions, or perhaps the information in the document is unclear.

Once you've categorized the information in the document, review your notes. The model helps you see where you have gaps. Whenever you find scant information in a given category, you know you need to ask more questions. Imagine you realize that you have a long list of the inputs that the system takes in but almost no notes on the output. You might want to ask detailed questions about how the system uses all those inputs or where users can see results.

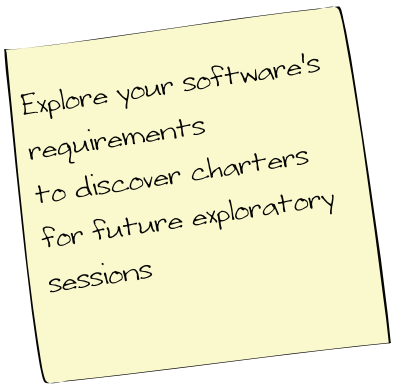
Draw a Model

As you are reading, look for passages that seem like the words would make more sense if expressed in a picture, perhaps as a state diagram (see Chapter 8, *Discover States and Transitions*, on page 75) or a context diagram (see Chapter 9, *Explore the Ecosystem*, on page 87). Interpreting a written description into a picture gives you another perspective on the software design.

On one project, sketching state models for the client and server sides in a proprietary protocol helped the entire team see the protocol in a new light. The team had originally been working from a prose description of the protocol. It turned out that the programmers working on the client had interpreted the protocol specifications slightly differently than the programmers working on the server, and there were latent bugs hiding in the cracks created by the subtle differences. A picture enabled the whole team to explore the interactions between the client and server more completely.

12.5 Practice Sessions

The best way to practice the concepts in this chapter is to analyze actual requirements.



- In your next requirements discussion, jot down test ideas using the heuristics from Appendix 2, *Test Heuristics Cheat Sheet*, on page 151. Share your test ideas with the other people in the room.
- Bring a stack of sticky notes to your next requirements meeting, and draft charters throughout the discussion.
- Find a requirements document or technical specification on which to practice active reading.

CHAPTER 13

Integrate Exploration Throughout

So far in this book, you've learned tactics for exploring—concrete, specific techniques for analyzing your target software, applying heuristics, and chartering your sessions.

It's time to turn our attention to larger questions of strategy: When should you explore? How does exploratory testing fit into the overall software development life cycle? How do you share and use the information that exploring yields? Who should do the exploring?

In this final chapter you'll find advice for integrating exploratory testing throughout your development cycle. You'll learn how to use the feedback from exploring to improve not just your software but also your development process. Finally, you'll discover how exploring can be a team activity.

13.1 Exploring as Part of Your Test Strategy

Section 1.1, *Two Sides of Testing*, on page 4, discussed how testing is really two things: checking and exploring. Checking the answers questions the extent to which the software does what you expect. Exploring uncovers risks, turning up answers to questions no one could have thought to ask in advance.

When your test strategy includes both checking and exploring and the team acts on the information that testing reveals, the result is incredibly high-quality software.

Real-Life Examples

Consider a story from Atomic Object in Grand Rapids, Michigan. The first time I visited the company, I was greeted by one of the programmers, Karlin Fox. Karlin took me over to a workstation and introduced me to a system that they had recently put into production. After a quick overview of the software, he suggested I explore it while he went to a meeting.

I started a recon session. Most of the time I can find several bugs in the first few minutes of using any software. Not this time. I dug deeper. Over the next half an hour I threw everything I had against the software. I could not find anything that I would call a bug. I experienced a strange mixture of chagrin and delight. I was chagrined that I couldn't find a way to break the software but delighted to see software that performed so well.

When Karlin returned I was bursting with curiosity. I asked about the process the team used.

I already knew that the team practiced Extreme Programming (XP). The programmers paired on writing code, so every line of code was peer reviewed before being checked in. They also test-drove everything, so they didn't write a line of production code until they had a failing test to demonstrate what that new code should do. And they integrated their work constantly, executing all those automated tests with every change.

However, even the best XP teams I'd seen up to that point hadn't achieved the level of quality I saw in this software. This team had done something additional.

Karlin grinned at me. Then he told me that they had an exploratory tester on the team. "He drove us nuts!" he reported. "Always finding stuff." Karlin paused to look at me. His smile widened. "It was great!" His grin turned sly. "There are bugs though," he said. He proceeded to show me the bugs I'd missed. They were subtle and negligible nits. I was even more impressed. By showing me the remaining defects, Karlin demonstrated exactly how high the team's standards were.

Atomic Object's story isn't an isolated incident. At Pivotal Labs in San Francisco, I worked on teams that used a process much like what Karlin described. We programmed in pairs, test-drove all our production code, practiced continuous integration, and also regularly explored. This process resulted in consistent high quality. One project in particular went incredibly smoothly. In the first six months after release, the software had just one bug that required an unscheduled maintenance patch in production.

What's more, that one bug only affected a small number of users and was easy to work around. On projects I had worked on a decade earlier, the team would have deemed that bug a minor issue and deferred the fix to some far-distant future release. However, our standards and the resulting quality of our software were so high that the bug stood out like a sore thumb. We fixed it right away. That project was over five years ago now. The software is still in production and running reliably.

Integrating Checking and Exploring Throughout Development

Both Atomic Object and Pivotal Labs use practices that come out of Extreme Programming (XP). These practices emphasize testing throughout development. In particular they emphasize these concepts:

- *Test-driven development (TDD)*: A programmer starts by writing an automated test that checks for the behavior she is about to implement. She executes the test, watches it fail, and only then does she write the production code to make the test pass.
- *Automated regression checks*: A side effect of practicing TDD is that the team has a fully automated set of unit-level checks. In addition, the team automates higher-level tests (end-to-end, or almost end-to-end, just bypassing the GUI) in order to check the core capabilities of the software.
- *Continuous integration (CI)*: At frequent intervals (usually just before checking in, but more often if necessary), each programmer working on the code base pulls all updates from the source control system, merges them in with his changes on his local workspace, and executes a set of unit tests that execute quickly (in seconds) and that provide coverage over the entire code base. This local test run tells him very quickly if his changes caused other parts of the code to break. He fixes anything that causes tests to fail prior to checking in.
- *Automated builds*: With each code check-in, the automated build server picks up the changes and executes the full suite of tests. These tests cover much more than the local unit tests that the programmer ran prior to check-in, and they naturally take longer to run. If the automated build fails for any reason, a team member (usually the last person to check in) stops to fix it right away. The team places a high value on keeping the code base clean and keeping the build green.

The cumulative result of all these practices is that the team checks its work each step of the way. All that checking means that the software is always ready to explore. The team knows that the software does what was intended (more or less), so team members are ready to tackle any risks and questions raised by exploration.

It's worth noting here that you can apply exploratory testing practices even on projects that don't use these disciplined engineering practices. However, my experience is that if your team combines rigorous checking with skilled exploration (and, of course, addresses any shortcomings these efforts reveal), the quality of your results is likely to be exceptionally good.

Exploring Early, Exploring Often

People give all kinds of rational-sounding reasons for holding off on exploring until closer to the end of a project. Some people think of exploratory testing as something to do if there's time left over toward the end of a release. Perhaps they believe that premature exploration will only reveal what they already know: that the software is not yet done. Or perhaps they think that it is not possible to explore before all the parts are in place. Some think that if they start exploring too early, there won't be enough to bother with exploring. Still others are concerned with having to retrace their steps. They argue that since they will just have to explore it all again later when it's done, starting too soon creates too much work.

None of these reasons are sufficient to justify delay. If you wait until the end of a development cycle to discover the surprises that exploration yields, it's too late. There's no time left to remedy the issues discovered. Furthermore, there's no time left to adjust the development process to fill in the systematic gaps that exploration may have revealed.

It is simply never too early to begin exploring. As discussed in Chapter 12, *Explore Requirements*, on page 121, you can start exploring in requirements discussions.

For that matter, the minute there is a line of code checked into the source control system, you can start exploring its behavior. You don't even need to wait for a user interface to be present.

On one of my first Agile projects, I started exploring a report feature before there was any way to enter the data to be displayed in the report. On previous projects I had been on, it would have been standard practice to delay testing until there was a user interface for entering data. However, that would have delayed my exploration by days if not weeks.

Because I know enough SQL to insert data into a relational database, I was able to exercise the reporting feature right away. In doing so, I found that the report had difficulty displaying certain types of data. The programmer was able to fix those issues early. Even more importantly, he was able to prevent other related defects. Early exploration is not just about finding and fixing bugs sooner. It's also about building quality in from the very beginning.

The bottom line is that exploring early and often leads to addressing risks sooner, and that leads to better software, fewer bad surprises, and greater overall predictability.

Dedicating Effort to Exploration

In many organizations that have a strong quality assurance or test department, there's an implicit assumption that exploring is the exclusive purview of a professional tester.

It is true that professional testers are likely to be better at exploring than their colleagues who have never studied testing. The core skills that make someone a good explorer neatly mirror the core skills of a good tester: analysis, observation, critical thinking, test design, and communication.

However, the key to success at both Atomic Object and Pivotal Labs involved including people with strong exploratory testing skills on the team from the very beginning. In neither case was there a separate team of testers or explorers working in isolation.

Karlin's team found success in having a dedicated exploratory tester. On my Pivotal Labs team, all of us were both programmers and testers. We explored our work at regular intervals. Further, all of us worked to hone our exploratory testing skills.

Your team might look more like Karlin's, with a dedicated explorer. Or your team might look more like mine, where everyone explores. Either way, exploring is part of the process from the very beginning and treated as a first-class activity rather than as an afterthought. So although you may or may not need dedicated explorers, you certainly need time dedicated both to exploring and to improving exploratory testing skills.

13.2 Paired Exploration

One way to get everyone on the team involved in exploring is to pair them up to explore. Pairing professional testers with others is particularly effective.

If a tester and a business analyst pair together, the tester is likely to learn more about the business and the expectations for the software while the business analyst is likely to learn more about how to design effective test cases. As a result of their conversations about expectations, the business analyst may even learn something about framing concrete requirements with examples.

If a tester and a programmer pair together on exploring, the programmer is likely to gain insight about risks while the tester gains insight about the architecture.

In one of my favorite examples, I was pair programming with another programmer who was a much better programmer than me but who was not as experienced in testing. We were writing code for a web application. I suggested

that we take a few moments to explore our implementation from an external perspective. I had a suspicion that our application was vulnerable to JavaScript injection.

Within a few seconds I had confirmed my suspicion: I succeeded in making the home page of the application pop up an alert box simply by inserting a JavaScript `alert` command into a text field in a user's profile.

I was pleased to have discovered the vulnerability before our first release, but my partner was puzzled. "Why would anyone do that?" he asked. I explained about cross-site scripting attacks and how this seemingly small vulnerability was a serious security risk.

My partner nodded, understanding. I suggested that we write some automated tests around the vulnerability we'd just found and then hunt down more vulnerabilities like it. My partner offered another suggestion. He proposed that we seed the test data we were already using with malicious JavaScript and then run our existing automated tests. Within a few minutes, we had found thirty or more places in the application that were similarly vulnerable. Within two hours, we had fixed all the vulnerabilities. We left the malicious JavaScript in our test data so that it would catch any new instances of related vulnerabilities.

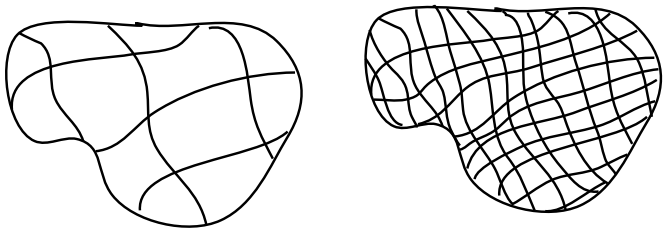
This story illustrates the power of pairing. As someone who has spent more time as a tester than as a programmer, I knew about an entire class of risks that my pair partner did not. As a more experienced and skilled programmer, my pair thought differently about test automation than I did. What took us a few hours of working together would have taken days if we had been working separately, and our final solution was more powerful and flexible than the automation I probably would have written if I'd been working alone.

So whenever someone asks me who should do exploratory testing, my response is that everyone explores. Perhaps you have professional testers on your team who do the bulk of the exploring. However, exploring is not restricted to those people. Programmers, business analysts, and anyone else with an interest in shipping a quality product have a hand in exploring.

13.3 Discovering Systemic Sources of Problems

Exploration obviously yields information about risks in the software under test. What may be less obvious is that exploratory testing can be a mechanism for assessing the process by which that software was made.

You may recall that the image in Section 1.1, *Two Sides of Testing*, on page 4 shows how you can imagine that your regression checks are a net. Here is that diagram again:



some nets have a loose weave

others have a tight weave

Whenever you discover a bad surprise when you are exploring that was not detected by previous tests, it suggests that there is a hole in the net.

some nets have big holes

When you fix the bug, you can also add a test to the net to patch the hole and make sure that bug, or a closely related cousin, cannot sneak back into the software.

If exploring yields a large number of bad surprises, it suggests that there is more than a hole in the net: the net is woven too loosely or it has many large holes. The net doesn't just need a small repair; the whole thing needs to be tightened up.

This is how exploratory testing can give you insight into the process that produced the software as well as into the software itself. If your exploration yields very few if any surprises, the process is working well. If, on the other hand, your exploration is uncovering seething masses of issues, it's not just the software that's broken; the process is broken too.

13.4 Estimating Exploration

One of the most common questions I'm asked is, "How can I estimate the amount of time to allocate to exploring?"

There is no one-size-fits-all answer to this question. The answer depends too much on the quality of the code base, the complexity of the system, and the number of variations you need to include in your exploration. You know far better than I do how much time you would need in order to explore your software to your satisfaction.

Usually when I offer this answer, the person questioning me nods in agreement. "That's right," the person says. "But how much time should I allocate…?"

One possible way to get a rough order of magnitude estimate on how much exploration you'll need for a given feature or area is to brainstorm a list of charters you would want to cover and then estimate the number of time-boxed sessions you would want to cover those charters.

The problem with estimating the effort needed to explore in this way is that it ignores the factors that are most likely to affect the actual time required to explore. Programming, checking, and exploring are inextricably intertwingled. Better development practices mean that you'll be able to explore more in less time. On the other hand, shortcuts leading to an unstable code base mean that both checking and exploration will take longer.

So rather than trying to estimate time for exploratory testing separately, I suggest providing a single estimate for the overall development effort. To do that, try the following:

1. Include the whole team—testers, programmers, and anyone else who will have a hand in the final deliverable—in the estimation discussion. No one can predict the future perfectly, but bringing together a wide variety of perspectives is likely to result in a better answer.
2. Ask everyone on the team to estimate the relative size of the overall effort. Common estimation approaches include using t-shirt sizes (small, medium, large) or the Fibonacci number sequence. (See Mike Cohn's book *Agile Estimating and Planning [Coh05]* for more ideas on estimation approaches.)
3. Offer the best estimate you can based on the information at your disposal; then reestimate as you learn more.

Teams that estimate together and then inspect and adapt usually find that their estimates improve over time.

However, it's also possible that you don't have an estimation problem at all. My colleague Dale Emery has observed that many teams that say they have an estimation problem actually have a negotiation problem. A manager asks for an estimate. The team says, "That will take four days." The manager makes a counteroffer, "Can you make it two?" This is no longer a discussion about estimates. It's a negotiation.

13.5 How to Tell When You Have Explored Enough

One of the reasons estimating your exploratory effort is so difficult is that exploration can be infinite. You can always try more variations. There are infinite possibilities. You can uncover more variables, analyze from more perspectives, and probe increasingly deeper. So it is useful to know when it is time to stop exploring.

Probably the most common criteria for stopping is that you run out of time. Unfortunately that means all too often that the software ships without anyone fully characterizing its capabilities, limitations, and risks. The team releases the software, but no one really knows what has been released. The predictable result is bad surprises in the field, frustrated users, and angry stakeholders.

So although time may be limited, running out of time is probably the worst criterion to use to determine when to stop exploring. Instead, consider using these guidelines to figure out when you've done enough exploration in a given area, at least for the moment:

- Your exploration has answered all the open questions, and you know of no more unknowns.
- You are not learning anything new. You have characterized the capabilities, limitations, and risks of the existing implementation. As a result, you feel like you are dealing with a known quantity with no more surprises to be uncovered, at least for now.
- More information will not change anything. You could gather more information, but there is no point. Your stakeholders have explicitly told you they do not need or want more information from exploring, and you know that no one will act on any information you find, including you.
- Other information has more value. Perhaps you could find more information in a given area, but there are other, more pressing questions to address in other areas or on other charters.

If you use these criteria to decide when to stop exploring and discover that you are not done exploring when the schedule says you are supposed to be done, it means the estimates were too low, not that exploring is taking too long.

13.6 Debriefing Stakeholders

Exploration does not lend itself to tidy status reports with test execution statistics. Counts of charters or test sessions cannot become the basis for pass/fail metrics because charters and sessions never pass or fail.

Yet you need some mechanism to report on progress. Your stakeholders have an interest in the information you're finding. Management has an interest in how you're spending your time. As a result, people often wonder how to report on the status of exploratory testing.

Reporting in the Daily Standup

Many teams have adopted the practice of a daily standup meeting as a means of coordinating their activities. In a classic standup, the three questions most commonly asked are these:

- What did you do yesterday?
- What are you planning to do today?
- What's getting in your way?

It might be tempting to say, "I explored. I plan to explore. Nothing is in my way." However that's no more useful to the team than a programmer saying, "I coded. I plan to code. Nothing is in my way." Even if you say, "I explored and filed three bugs," it's still not much detail. The team needs more information in order to coordinate effectively.

So instead of answering the three questions with a literal report of what you did, consider how you could use that time to provide the team with headlines about the information you're discovering. Reframe the questions in your head a little before you answer:

- What is the most interesting thing you discovered about the software while exploring yesterday?
- What risks will you be exploring for today?
- Is there anything that you need in order to use your time more effectively? That could include information, changes to improve testability, resources like test environments, or even just feedback on whether or not the team values the information you are discovering.

So for example, a standup report might be something like this:

"Yesterday, I explored the import function using malformed data. I discovered that if any import file is empty, it causes the process to die. The rest of the files in the batch get thrown out. I filed a bug on that. I plan to continue exploring malformed data today. I suspect that there are other, related problems. Before I do that though, I'd like to know if the team would value more bugs like that one."

Debriefing Decision Makers

At critical decision points you may need to debrief the key decision makers. They're trying to answer difficult questions: Is the software ready for production yet, or is there more to do before release? Does the software meet our quality criteria? Is the automated test coverage sufficient or does it need to be increased? Is the software stable enough to put into beta test?

Stakeholders typically appreciate brevity. They will not have the patience to sit through a rambling narrative. So you need to prepare to give a succinct and distilled view into the state of the software: its current capabilities, limitations, and risks. You'll need to address the following in your debrief:

- What have you explored so far?
- What have you found?
- What remains to be covered? Put another way, what are the remaining unknowns?

These three questions form the template for a short summary on the results of exploring.

When answering the question of what you have explored so far, give a high-level view of the areas or charters you have explored and a rough indication of your depth of coverage. For example, you might say something like "I have explored each of the major functional areas plus the interaction between the new authentication scheme and each of the functions that involve authentication. I have spent most of my time focusing on the web interface and have spent significantly less time on the API."

When you discuss what you've found, mention crucial bugs, but also discuss what you've observed to be working as expected. For example, "I have not found any major differences in the behavior of the system with the new authentication scheme. I did find two severe bugs that I reported and that have already been fixed related to sessions not timing out correctly."

Finally, be clear about what remains to be explored and what possible risks remain to be uncovered. Knowing where there is an absence of information could make a difference in the decisions your stakeholders make. For example, "Because I was more focused on the web interface than on the API, the API has not been covered as well. In particular, since there were issues related to session timeouts with the web interface, there could well be issues with session timeouts in the API."

Throughout your discussion, remember to focus on externally verifiable facts. You are an advisor, but beware of offering your own judgment as a conclusion. There is a world of difference between saying "It's not ready yet" and "While exploring, I found the following major blocking defects..." The former is a conclusion, a judgment. The latter is the information that you used to form the conclusion. The decision makers need to know your evidence so that they can come to their own conclusions.

As you are reporting your findings, be alert for feedback. You will learn a world about the kind of information that your stakeholders value if you listen carefully for clues about what they most appreciated learning. If you reported a bug and they responded, "Oooh! Good catch!" that suggests they might like more information like that. On the other hand, if you hear "That's not surprising. It doesn't work the way you think it does," it means that not only do they not value the information you're providing, they feel like it's a waste of your time and theirs.

Of course, if you think that it would be negligent for you not to look for a given kind of risk, you should say so. Explain why you think that category of risk is important. Explain the consequences of ignoring that kind of risk. Put it in concrete and realistic terms.

13.7 Capturing Useful Nuggets of Wisdom

It is inevitable that in exploring you will learn something that isn't strictly information about the behavior of the system. All too often such information ends up buried in a notebook and not shared.

On one project, I used a command-line SQL interpreter to query the database behind the scenes to get access to the system's data that wasn't displayed through the user interface. I had to ask the developers a lot of questions to figure out how to get at the database, and even then it took me a couple of hours of trial and error to figure out how to get the information I most needed out of it. However I chalked up my stumbles to the fact that I was new on the

project. I figured that everyone else on the project already knew how to peer into the database to see what was really going on.

Imagine my surprise a week later when I learned that one of the developers (one of the very people I had quizzed the previous week) was struggling to do the same thing I had done. He knew part of the trick to getting to the database, but he didn't know everything I had learned.

If I had captured what I'd learned about querying the database on the team wiki, I would have saved the developer some time. As it was, as soon as I found out what he was trying to do, I was able to show him exactly what he needed. I then captured the information on the wiki for future reference.

Some of the most valuable information that explorers find is how to explore effectively. Make a point of capturing that information for other team members to use.

13.8 Practice Sessions

Practice debriefing. Imagine that you've just stepped into an elevator with an executive in your company. She asks, "So, how's it looking?" It's a short elevator ride, just a floor or two. You have sixty seconds to give her an honest and balanced report. What would you say?

Find something you know that others don't, and document it. Perhaps you know how to set up some critical aspect of the test environment. Maybe you know where to snag production-like data. Or maybe it's something as simple as a scripting trick to generate realistic data for exploring. Whatever it is, there is certainly something that you know that others on your team would value. Figure out what that nugget of wisdom is and capture it in your team's central information repository.

Finally, revisit Section 1.4, *Practice Sessions*, on page 8, with one or more of your team members. Consider the questions that your test strategy is designed to answer and how you can integrate more, earlier exploration into your overall process to answer those questions more effectively.

APPENDIX 1

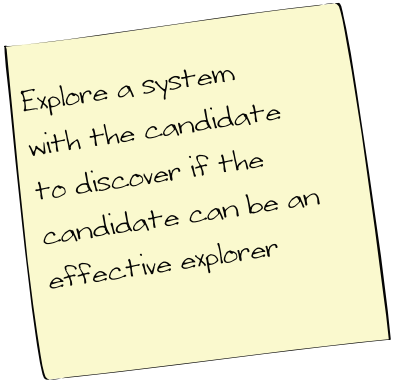
Interviewing for Exploratory Testing Skills

As a hiring manager, I have interviewed hundreds of testers and hired dozens.

Interviewing and hiring testers is always an interesting challenge. The work a tester does is not tangible. Certainly, you can read a tester's bug reports or test scripts. However, neither artifact tells you much about how well the tester will perform at actual testing. The difference between adequate testers and stellar testers rests in their approach to analyzing and investigating software and systems.

I've found that the best way to gauge candidates' skills is to have them test something. Of course, that makes sense. Asking people to demonstrate that they can do the work you are considering hiring them for is a time-honored interview practice.

However, I take that idea a step further and think of an interview as an exploratory session with a special kind of charter:



Note that although I developed the interview practices in this appendix when I was hiring testers, the idea applies to hiring into any role where the ability to explore is useful: programmers, product managers, and so on.

A1.1 Paired Exploratory Testing as Part of an Interview

In the interview, your goal is to assess the candidate's ability to explore effectively:

1. Can the candidate explore systematically to uncover vulnerabilities and surprises?
2. Can the candidate effectively communicate what was discovered?
3. Does the candidate adjust the kind of information sought based on feedback?

To structure an effective exploratory interview session, here are the steps I take.

Setup

If possible, it's best to do this in person so that you can pick up on subtle cues about the candidate's comfort level. Creating safety is important, so if the candidate seems uncomfortable, you can offer to take a break.

Choose something to test. It might be your real production software or it might be a toy demo system. Whatever you choose should be something representative of the kind of thing that the candidate would be exploring in the real world. If you ask candidates to explore a hand-held toy but in the real world they will be testing a web application, you won't find out much about their ability to exercise web applications.

Set the candidate's expectations: "We're going to explore together for forty-five minutes to an hour."

Starting

Position yourself at the keyboard. As the host, you'll drive until the candidate asks for control.

Tell the candidate, "I'll drive for a while so you can focus on thinking about the system. You can have control of the keyboard whenever you want. Just ask."

Establish the charter with the candidate. Usually a recon charter is a good way to start: "Let's explore this system to discover how it works."

Ask the candidate to start by suggesting things to try or by asking questions.

A good candidate will ask probing questions, such as: "What's that? How does that work? How does this interact with that? What happens if we…?" and "How can we…?"

Observing the Candidate

As the candidates direct you in exploring the system, notice their patterns of thinking. Are they seeking information beyond the obvious? Are they using test heuristics or test design techniques to guide their actions? Are they exploring with a theory of error to find particular types of bugs? Are they staying on charter?

As you explore, let the candidates go wherever they wish. If you direct the exploration, you won't know what they would have done if left to their own devices.

Debriefing

Continue exploring until you sense that the candidate has a grasp on how the software operates or for fifteen minutes, whichever happens first.

Pause the exploration. Take your hands off the keyboard and face the candidate. Say, "Let's pause our exploring for a moment to talk about what you've learned so far."

Then ask, "What can you tell me about this system so far? What have you observed that you think is interesting or surprising?"

Listen carefully to the answer. This will give you a great deal of information about how effectively the candidate communicates, and it will also tell you a lot about what the candidate thinks is significant.

Remember that the goal was not to find bugs, and the candidate probably doesn't know much about what to expect from the system under test. So at this point it's OK if the candidate didn't notice any bugs. It's also OK if the candidate has a long list of possible bugs. Bugs aren't the point.

At this point you might know enough to cut the interview short. If candidates seem lost at this point, more exploration probably will not help them find their groove.

Steering

If the candidate is doing well enough, continue for another mini session. Choose a different charter designed to reveal whether the candidate can investigate a system to look for a specific type of information.

For example, if the system you are exploring is rife with states and transitions, you might choose a charter about discovering vulnerabilities related to interrupts or timing. If the system you are exploring is data intensive, you might choose a charter about discovering vulnerabilities related to bad inputs. If the system involves graphic outputs, you might choose a charter about discovering problems with rendering.

Whatever you choose, it should be something specific and something that is representative of the exploring you expect the candidate to be doing on the job.

Ask the candidate to continue exploring. Offer the candidate the controls. And ask each candidate to think out loud while driving.

Now you're listening for all three core skills: the ability to explore systematically, the ability to articulate what was found clearly, and the ability to focus on finding the information requested.

The second debrief will look much like the first: ask the candidate to characterize the capabilities and limitations of the system discovered so far.

In addition, ask the candidate, "If we were to do a third session, tell me what you would want to investigate and why." This will demonstrate the candidate's ability to direct his or her own explorations.

A1.2 Assessing the Candidates

You have a good candidate on your hands if he or she exhibits curiosity, asks good questions, demonstrates analytical and critical thinking skills, and articulates findings. If this person also happens to have technical skills that will enable him or her to get a running start, so much the better. However, remember that someone who exhibits curiosity and an ability to learn will be able to pick up on details quickly.

APPENDIX 2

Test Heuristics Cheat Sheet

This appendix gathers all the test design heuristics scattered through the book. It also describes additional heuristics not covered earlier but that you may find useful as you progress further in your explorations.

A2.1 General Heuristics

These heuristics can apply to absolutely any kind of system with any kind of interface.

Abstract

Remove details in a model. Simplify by omitting low-level, granular bits so that the model focuses less on specifics and more on concepts.

For example, with an interaction diagram you could replace details about a set of communications with a single line named for the essential interaction that the communications represent. If you were working on an email client, the steps that the client goes through to send an email include connecting to the server, authenticating, and transmitting the data. Instead of identifying each of these steps in your model, you can remove the details and raise the level of abstraction by simply labeling the entire client/server interaction as "Send Email."

Never and Always

The things that the software should always do (such as ensure that accounts balance) or never do (such as destroy user data). To discover the Nevers and Alwayses for your system, talk with subject matter experts and business stakeholders.

Beginning, Middle, End

Vary the position of an element. You could vary where a given element appears in a sequence, or you could act on an element in a given position. Examples of applying this heuristic include the following:

- Deleting an item from the beginning, middle, or end of a list
- Pasting text at the beginning, middle, or end of a line
- Placing a special character at the beginning, middle, or end of a value
- Performing an action at the beginning, middle, or end of a sequence

Watch for indications that position makes a difference. Also be alert for bad behavior, such as data appended at the end being truncated or data inserted at the beginning or in the middle causing other contents to be truncated.

Centralize Everything

If things are scattered, subdivided, or organized in separate bins, put them all in one place. Examples:

- *For software where items can be organized into folders*: Take a large collection of folders with items in them and move everything into a single folder or a single deep hierarchy of folders.
- *For software where ownership of an item can be transferred between accounts*: Transfer ownership of a large number of things from varying accounts into a single account.

Change the Model

Take a model of the system you're testing and translate it into a different representation of the model. Examples:

- Change a state diagram with circles and arrows representing the states and transitions into a table showing states as columns, events as rows, and resulting states as data in the cells.
- Change a linear outline into a mindmap, or a mindmap into a linear outline.

CRUD

Create. Read. Update. Delete. These are the essential actions you can take on any data element within a system. Use CRUD in conjunction with other heuristics. For example:

- *With Beginning, Middle, End*: Create an element at the beginning of a list, in the middle of a list, or at the end of a list.

- *With Goldilocks*: Update a data element with a value that is too small, too big, or just right.
- *With Zero, One, Many*: Delete an item with no child records, with one child record, or with many child records.

Decentralize Everything

If things are all in one place, scatter them, subdivide them, or organize them into separate bins. Also, make connections across distributed systems. Examples:

- *For software where items can be organized into folders*: Create a large number of folders, each with a few items.
- *For software that can be configured to work in a distributed environment*: Configure the system so that each part exists on its own separate machine and separated from the others with subnets or firewalls.

Follow the Data

Perform a sequence of actions involving data, verifying the data integrity at each step. This is appropriate for any system where users can perform activities on data like entering, searching, reporting, importing, exporting, updating, and viewing. Following the data is particularly useful when combined with other data-related heuristics. For example:

- *With data that has special characters, including SQL injection or JavaScript injection attacks*: Import it. Search on it. View it. Run reports that include it. Export it.
- Create data that violates domain-specific rules and then import it. Sometimes the logic that prevents bad data from being entered directly is missing from batch import utilities.
- *With Zero, One, Many*: Perform data-centric actions such as running reports or exporting data with entities that have no child records, one child record, or many child records.

Goldilocks

Too big, too small, or just right. This can apply to anything that has a valid range of sizes, such as a range of dates, a numerical value, the length of a string or text, or the size of a file. Behavior to watch for includes the following:

- Unhelpful error messages, such as raw exceptions with stack traces or mysterious error codes

- Truncating large data without notice
- Failing to save data without notice

Interrupt

Find ways to interrupt key processes, such as logging off, shutting down the machine, rebooting the machine, killing the process from the operating system, disconnecting the machine from the network, forcing the machine into hibernation, letting the session time out, and cancelling the operation. Watch for misbehavior, such as the following:

- Unhelpful error messages, such as raw exceptions with stack traces or mysterious error codes
- Inability to start the process (ever) again
- Unrecoverable data loss or corruption

Reverse

Do things in reverse order. Undo everything. Skip to the end and work backward. Examples:

- *For software with an Undo feature*: Perform a wide variety of actions, then undo all your work step by step.
- *For software with a defined workflow*: Accept all the default values to get to the end of the workflow; then use the back button to go back to each screen and change the values.

Some, None, All

Some of the items in a set, none of them, all of them. Related to 0, 1, Many but for defined sets of items or attributes such as permissions, configuration options, tags, multiselect fields, and checkboxes. Behavior to watch for includes the following:

- None treated the same as All—for example, the system treating a user with no permissions the same as a superuser with all permissions
- Problems with count or percentage calculations, including divide-by-zero and off-by-one errors
- Display issues, particularly with None or All
- Data truncation with All

Starve

Starve the program of any required resource, including CPU, memory, network bandwidth, or disk space. Watch for these responses:

- Unrecoverable data loss or corruption
- Unhelpful error messages, such as raw exceptions with stack traces or mysterious error codes
- Unexpected program or session termination

Too Few

Applies when you have counts of things. Create conditions with fewer things than the software expected. For example, have too few items to populate a list or too few line items on an invoice.

Too Many

Applies when you have counts of things. Create conditions with too many things, such as too many active sessions or connections.

Useful Approximations

Especially if you are dealing with a complex domain, it may be impossible to determine exactly what the expected result is for a given experiment. In that case, you can use an approximation to assess correctness. You can evaluate results against a range of possible values to see if it falls inside the range. You might also check the characteristics of the results, such as whether the computed values are increasing or decreasing.

Violate Data Format Rules

The software expects data to conform to rules. Ages should never be negative. IP addresses should always contain four numbers ranging from 0 to 255. Email addresses should always have a username separated from the domain by an @ sign. Custom file formats have a specific structure. Try violating these rules and see how gracefully (or not) the software handles the badly formatted data.

Zero

Cem Kaner's law of zero is that if there is a zero, something will attempt to divide by it. It also turns out that when software is expecting a set of things, it frequently does not handle the empty set (zero elements) well.

Zero, One, Many

Zero things, one thing, many things. The things you are counting might include results returned in a search, characters in a string, bytes in a data stream, descendants in a hierarchy, a number of files, lines within a file, accounts, activities in a log, or any domain-specific object or attribute. Behavior to watch for includes these issues:

- Problems with plurals, such as "0 record found" or "1 records found"

- Problems with count or percentage calculations, including divide-by-zero and off-by-one errors
- Indications of performance issues as the count of things increases

Zoom

Focus on the concrete details, narrowing your field of vision if necessary.

For example, with a state model, you could identify an event in your model and identify substates by breaking down the steps the software takes in transitioning from one state to the next. For instance, with software that allows users to create and update items, you could take a Save event and break it down into steps such as transmitting data, validating data, and responding. Each of these steps is a state with events that trigger the transition between them.

A2.2 Web Heuristics

These heuristics apply specifically to web-based applications.

Back, Forward, History

Users can navigate through a web application by interacting with the web application or by using the history features in the browser, including the back and forward buttons as well as the history. Rich web applications don't always handle this well. Watch for the following:

- Warnings about POST data being re-sent to the server
- Duplicate transactions
- 404 errors
- Pages displaying with only partial data
- Errors, including broken images and broken links

Bookmark It

Users bookmark all kinds of pages, including pages that might not be designed to be bookmarked. In particular, try bookmarking pages that appear in the middle of a sequence, such as the checkout page in a purchase flow. Watch for the same kinds of problems as with Back, Forward, History.

Bibliography

[Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, New York, 1990.

[Coh05] Mike Cohn. *Agile Estimating and Planning*. Prentice Hall, Englewood Cliffs, NJ, 2005.

[Coo99] Alan Cooper. *The Inmates Are Running the Asylum*. Sams Publishing, Indianapolis, IN, 1999.

[Cop04] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, Boston, Massachusetts, 2004.

[Fey97] Richard Feynman. *Surely You're Joking, Mr. Feynman!*. W. W. Norton & Company, New York, NY, 1997.

[KBP02] Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing: A Context-Driven Approach*. John Wiley & Sons, New York, NY, 2002.

[Kan88] Cem Kaner. *Testing Computer Software, First Edition*. Tab Books, Blue Ridge Summit, PA, 1988.

[Lev95] Nancy Leveson. *Safeware*. Addison-Wesley Professional, Boston, MA, 1995.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, New York, NY, 1979.

[Sla89] Robert Slater. *Portraits in Silicon*. MIT Press, Cambridge, MA, 1989.

[Wir02] Rebecca Wirfs-Brock. *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley Professional, Boston, MA, 2002.

Index

A

Abstract heuristic, 151
Activity Monitor, 30
alarm clock example, 82–84
Always rules, 48–50
API, testing, 34, 100–102
approximations of correct results, 52–55
Ariane 5 rocket example, 37
Atomic Object example, 133–134, 137
attributes of entities, 67–68
automated builds, 135
automated regression checks, 135

B

Back, Forward, History heuristic, 156
batch processing system example, 99
Beginning, Middle, End heuristic, 39–40, 152
billing system example, 115
bird flight speed example, 53
Bookmark It heuristic, 156
bug database, as input to charter, 21
bugs
  characterizing, 106–107
  unreproducible, researching, 117–119
builds, automated, 135

C

Centralize Everything heuristic, 152
Change the Model heuristic, 152
charter for recon session, 110–112
charter for tests, 11–24
  existing artifacts as input to, 21
  implicit expectations for, 19
  revision of, during testing, 21, 24, 148, 150
  risks as input to, 21–23, 50, 110
  software requirements as input to, 17–18, 128–130
  specificity of, 15–16
  stakeholders' input to, 16, 19–20, 50
  template for, 13–15
CI (continuous integration), 135
communication protocol standards, 51
communication skills, 148–149
consoles, 31–32
context diagram, 88
continuous integration (CI), 135
core capabilities of software, 48–49, 125–126
correctness of results, assessing, 47–55
credit card payment system example, 88–91
CRUD (Create, Read, Update, Delete) heuristic, 70–73, 152

D

data formats, changing, 41
databases, monitoring, 30
Decentralize Everything heuristic, 153
deployment diagram, 88
designing tests, *see also* charter for tests
  heuristics for, 151–156
  variables in, 6, 33–44
development
  exploratory testing during, 133–138
  systemic problems with, discovering, 138–139

E

Easy Button video, 111
ecosystem, 87–96
  determining, in recon session, 110
  diagram of, 87–92, 131
  external dependencies in, 89
  Follow the Data heuristic for, 95
  internal dependencies in, 90–91
  testing based on, 93–95
  trust boundaries of, 92–93
  user interfaces in, 88
email client example, 60–62

entities, 67–73
attributes of, 67–68
CRUD heuristic for, 70–73
Follow the Data heuristic for, 73
relationships between, 69–70
environment, *see* ecosystem
ERD (entity relationship diagram), 69–70
evaluating test results, *see* learning from tests
events, 77–78, 80–81
executing tests, 7, 136
existing systems, testing, 109–119
documenting findings of, 116
multiple testers for, 112–114
recon session for, 109–112
stakeholders, interviewing, 114–116
techniques for, determining, 116
unreproducible bugs, researching, 117–119
exploratory testing, 4–8, 11
charter for, 11–24
information learned from, documenting, 144–145
integrating with development, 133–138
process for, 6–7
results of, reporting to stakeholders, 142–144
skills required for, interviewing for, 147–150
skills required for, observational, 25–30
strategy for, 4
time required for, estimating, 140–141
time-boxed sessions for, 7
when complete, determining, 141–142
when to begin, 7, 136
who should perform, 137–138
external dependencies, 89
externally generated events, 77
Extreme Programming (XP), 134–135

F

file locations, changing, 40
filesystem, monitoring, 30
Follow the Data heuristic, 73, 95, 153
form fields, changing, 34
format of data, changing, 41

G

Galactic Modeling Language (GML), 88
geographic locations, changing, 40
GML (Galactic Modeling Language), 88
Goldilocks heuristic, 153

H

Health Insurance Portability and Accountability Act (HIPAA) standards, 51
heuristics, 151–156
Abstract, 151
Back, Forward, History, 156
Beginning, Middle, End, 39–40, 152
Bookmark It, 156
Centralize Everything, 152
Change the Model, 152
CRUD (Create, Read, Update, Delete), 70–73, 152
Decentralize Everything, 153
Follow the Data, 73, 95, 153
Goldilocks, 153
Interrupt, 154
Never and Always, 151
Reverse, 154
Some, None, All, 154
Starve, 154
Too Few, 155
Too Many, 155
Useful Approximations, 155
Violate Data Format Rules, 155
Zero, 155
Zero, One, Many, 155
Zoom, 156
HIPAA (Health Insurance Portability and Accountability Act) standards, 51

I

inattentional blindness, 26
indirectly accessible variables, 36
Inputs, Processing, Outputs model, 131
installation test example, 27–28
internal dependencies, 90–91
Interrupt heuristic, 154
interviewing for exploratory testing skills, 147–150
inventory management system example, 69–70

L

learning from tests, 7
correctness of results, assessing, 47–55
documenting information learned, 144–145
observation of unexpected results, 25–30
reporting results to stakeholders, 142–144
Lewis and Clark's expedition, 11–12
login screen example, 76, 78
logs, 31–32

M

many-to-many relationship, 70
Mars rover example, 37–38
monitoring
consoles for, 31–32
databases, 30
filesystem, 30
logs for, 31–32
network traffic, 30
system resources, 30
web server traffic, 30
moonwalking bear example, 25–26
multivoting, 23

N

network traffic, monitoring, 30
Never and Always heuristic, 151
Never rules, 48–50

Nightmare Headline Game, 21–23
nouns and verbs, creating random actions using, 60–62

O

observational skills, importance of, 25–30
obvious variables, 34–35
one-to-many relationship, 69
Open Web Application Security Project (OWASP) guidelines, 52
operating system, monitoring, 30
oracle problem, 53
OWASP (Open Web Application Security Project) guidelines, 52

P

paired exploration, 137
personas, 63–64
Pivotal Labs example, 134, 137
position, testing variations in, 39–40
preference settings, users changing, 35
Process Monitor, 30
programming language, testing, 102–104

Q

quality criteria for software, 49
queuing theory example, 54

R

random number generator example, 54
recon session, 109–112
regression checks, automated, 135
relationships between entities, 69–70
requirements, *see* software requirements
Reverse heuristic, 154
risks, determining, 21–23, 50, 110
routing algorithm example, 54

S

Sarbanes-Oxley standards, 51
Scrabble Flash game example, 112
sequences, varying, 59–62
software requirements, 121–132
  for comparable software, 52
  core capabilities, 48–49, 125–126
  documented, analyzing, 130–132
  expectations for, conflicting, 127–128
  as input to charter, 17–18, 128–130
  Inputs, Processing, Outputs model for, 131
  internal consistency, 50–51
  Never and Always rules for, 48–50
  quality criteria, 49
  standards, adherence to, 51–52
  testers' involvement in, 121–128
software, existing, *see* existing systems, testing
Some, None, All heuristic, 154
sorting function example, 102–104
source code, as input to charter, 21
square root function example, 54
stakeholders
  reporting testing results to, 142–144
  role in exploring existing systems, 114–116
  role in test requirements, 16, 19–20, 50
standards, adherence to, 51–52
Starve heuristic, 154
states, 75–85
  detecting, 76–77
  events triggering transitions of, 77–78, 80–81
  interrupting, 81
  model of, 78–80, 131
  table of, 82–85
  testing based on, 80–82
  for unreproducible bugs, determining, 119
steering, 7, 150, *see also* learning from tests
string comparison example, 100–102
subtle variables, 35
  examples of, 36–38
  identifying, 38–44
support calls, as input to charter, 21
system resources, monitoring, 30
system-generated events, 78

T

TDD (test-driven development), 135
testability hooks, 31
testing
  coverage of, 4–5
  exploratory, 4–8, 11
  types of, 3, 5–6
Therac-25 case, 36–37
time estimates, 140–141
time zones, changing, 40
time-boxed sessions, 7
time-related events, 78
Too Few heuristic, 155
Too Many heuristic, 155
transitions of state, events triggering, 77–78, 80–81
trust boundaries, 92–93

U

unexpected noise example, 29–30
unexpected results
  ability to observe, 25–30
  testing for, 30–32
unreproducible bugs, 117–119
URL parameters example, 35
Useful Approximations heuristic, 155
user interface
  in ecosystem, diagramming, 88
  guidelines for, 51
  randomizing navigation through, 62

users
actions of, as events, 77–78
actions of, unpredictable, 59
personas for, 63–64

V

variables in tests, 6, 33–44
with CRUD heuristic, 71–73
determining, in recon session, 110
indirectly accessible, 36
infinite number of, 33, 44
obvious variables, 34–35
sequence of interactions, 59–62
for state models, 82
subtle variables, 35
for unreproducible bugs, determining, 119
user personas as, 63–64
verbs and nouns, creating random actions using, 60–62
Violate Data Format Rules heuristic, 155

W

W3C standards, 51
web service
performance example, 106–107
testing, 104–106
web-based software
credit card payment system example, 88–91
heuristics for, 156
monitoring, 30
navigation of, 59
standards for, 51
What If? game, 93–95, 126–127

X

XML web service example, 104–106
XP (Extreme Programming), 134–135

Z

Zero heuristic, 155
Zero, One, Many heuristic, 155
Zoom heuristic, 156

Behaviour-Driven Development & Testing

Go farther with automated testing.

Be Agile

Don't just "do" agile; you want to *be* agile. We'll show you how.

Make it Work

Do retrospectives the right way, and see how to get new ideas accepted.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
http://pragprog.com/book/ehxta
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *http://pragprog.com/titles/ehxta*

Contact Us

Online Orders:	*http://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://pragprog.com/write-for-us*
Or Call:	+1 800-699-7764